Sistemas Informáticos
Curso 2007 - 2008

Proyecto <e-Adventure 3D>:

Entorno de Autoría Para la Creación de Aventuras 3D Educativas en Entornos Virtuales de Enseñanza

F. Javier Torrente Vigil
Guillermo Cañizal Alzola
Ángel del Blanco Aguado

Dirigido por:
Baltasar Fernández Manjón
Pablo Moreno-Ger

Facultad de Informática
Universidad Complutense de Madrid

<e-Adventure 3D>

Proyecto de Sistemas Informáticos
Facultad de Informática

Universidad Complutense de Madrid

Autores:
Francisco Javier Torrente Vigil
Guillermo Cañizal Alzola
Ángel del Blanco Aguado

Profesores directores:
Dr. Baltasar Fernández Manjón
Dr. Pablo Moreno Ger

Curso 2007 / 2008

Agradecimientos

Este trabajo ha sido el fruto de muchas horas de dedicación e ilusión. Desde que comenzamos allá por agosto de 2007 a trastear con el motor Java Monkey Engine, entonces desconocido para nosotros, ha transcurrido casi un año de duro trabajo. Sin embargo, como en casi todo esto no es únicamente el resultado de sus autores.

En primer lugar debemos agradecer el apoyo a nuestras familias por el apoyo recibido. A nuestros amigos por aguantar durante horas conversaciones sobre Java, e-Learning, vectores y demás temas frikis con un aguante cercano al masoquismo (especialmente Jessica). También a Roberto Tornero por habernos ayudado con sus gráficos para que <e-Adventure3D> tenga un mejor aspecto.

En segundo lugar, A Bruno, Francis y Edu, cuyo <e-Adventure> ha sido una fuente de buenas ideas de diseño de la aplicación.

Por último (aunque no por ello menos merecido) a los Drs. Pablo Moreno Ger (cuya tesis doctoral ha sido la inspiración del proyecto) y Baltasar Fernández Manjón por su apoyo y orientación.

Se autoriza a la Universidad Complutense de Madrid a difundir y utilizar con fines académicos, no comerciales y mencionando expresamente a sus autores, tanto la memoria como el código, la documentación y/o el prototipo desarrollado.

F. Javier Torrente Vigil Guillermo Cañizal Alzola Ángel del Blanco Aguado

Palabras clave para búsqueda bibliográfica:

- E-Learning
- Educativo
- Videojuego
- 3D
- LMS
- Adaptación
- Evaluación
- Editor
- Gráficos
- Motor

Table of Contents

Chapter I

Introduction

1. Sobre <e-Adventure3D>

Tanto el aprendizaje como los videojuegos tienen características similares. No sólo ambas son tareas desafiantes, sino que también fuerzan al estudiante (o jugador) a alcanzar un profundo entendimiento del campo de estudio, a habituarse a él e incluso llegar a interiorizarlo. Además la efectividad de usar videojuegos en entornos educativos ha sido probada ya que disminuye la frustración de los alumnos, aumenta su estimulación, promueve el aprendizaje colaborativo y fomenta su motivación. Probablemente, éstas son las principales razones para aplicar video juegos a la educación.

Sin embargo, combinar el aprendizaje y los videojuegos no es sencillo. El desarrollo de videojuegos es una tarea a tiempo completo que requiere conocimientos técnicos avanzados y de programación. Muy pocos proyectos educativos podrían afrontar el gasto de un video juego. Tal elevado coste ha sido señalado como la principal limitación para los videojuegos educativos.

De ahí surge <e-Adventure3D>, quién recoge el testigo de su predecesor, <e-Adventure> (de ahora en adelante <e-Adventure2D>). El proyecto trata de aportar una plataforma intuitiva pero completa y potente para el desarrollo de aventuras 3D con fines educativos con un coste razonable. Con tal propósito la plataforma consta de dos aplicaciones, la *herramienta de edición orientada a autor* y el *motor de juego*. El diseñador del juego (probablemente un profesor o instructor) toma los recursos producidos por los artistas (modelos 3D, música, imágenes, etc.) y desarrolla el juego con el editor. Por su lado el motor de juego es el encargado de ejecutar dichos juegos. Los educadores no necesitan tener ningún conocimiento técnico sobre el desarrollo de videojuegos, simplemente han de centrarse en los aspectos educativos. Por esta razón la plataforma <e-Adventure3D> incluye características para la evaluación automática del alumno (conocido como “assessment”), así como herramientas para ajustar el comportamiento de los juegos dependiendo de distintos factores, en lo que se conoce como “adaptation” (no todos los estudiantes tienen la misma forma de aprender).

Por último, simplemente remarcar la naturaleza innovadora del proyecto. Aunque se han llevado a cabo algunas experiencias satisfactorias aplicando videojuegos en entornos educativos, existen pocas plataformas que soporten completamente el ciclo de desarrollo que aquí se propone (Adventure Maker, Game Maker), y ninguna de ellas en 3D. Por ello éste proyecto supone un avance radical y un desafío en el campo de los videojuegos educativos.

2. About <e-Adventure3D>

Both learning and playing video games have plenty of features in common. Not only are both challenging tasks, but also force the learner (or player) to reach a deep understanding of the specific field, get used to it, and even internalize it. In addition, the effectiveness of using video games in educational environments has been proved since it decreases the frustration of the students, increases their stimulation, promote collaboration and enhances the motivational aspect. These are, maybe, the foremost motivations to apply video games in education environments.

But to join learning and video games together is not easy. Game developing is a full-time-consuming activity, and also requires advanced programming skills (much further advanced when talking of 3d games). Very few educational budgets could afford the expenditure of a video game. Such high cost of the development of video games has been pointed out as the principal limitation for their introduction in educational contexts.

Hence arises <e-Adventure3D>, which picks up the baton from its predecessor, <e-Adventure> (from now on <e-Adventure2D>). It aims to provide an intuitive but complete and powerful platform for the development of educational 3D adventure games at a reasonable cost. For that purpose the platform is composed of two applications, the *author editor tool* and the *game engine*. The adventure designer (likely a teacher or instructor) takes the assets (3D models, audio tracks, image files, etc.) produced by the artists and uses the user-friendly editor to develop the game. Then it is ready to be run on the <e-Adventure3D> engine, the application which executes the games produced by the editor. Instructors do not need to have technical background, just focus on the story of the game and the educational traits. For that reason the <e-Adventure3D> platform includes features for the automatic evaluation of the student (formerly known as assessment), and adaptation tools to improve its "educational efficiency", gauging the behaviour of the games depending on diverse issues.

As a last consideration, just remark the innovative nature of this project. Although there have been some satisfactory experiences with video games in education, very few platforms have been developed to support the full development cycle here proposed (Adventure Maker, Game Maker), and any of those in 3D. Therefore, this project involves a radical and challenging advance in the arena of educational video games.

3. Sobre este documento

Bienvenido a la plataforma <e-Adventure3D> y gracias por tomar su tiempo en leer este documento. Antes de nada nos gustaría reseñar algunos aspectos acerca de la estructura y contenido del mismo.

En primer lugar, el documento se encuentra organizado en capítulos de forma cronológica (los primeros capítulos describen planteamientos iniciales del proyecto, los del cuerpo medio las características del mismo y características de su propio desarrollo y los últimos tests y conclusiones). La mayor parte de los mismos han sido escritos en inglés para facilitar el uso de la plataforma en organismos y universidades extranjeras, salvo algunos apéndices y el caso de estudio que por falta de tiempo fueron escritos en español. Pedimos disculpas por las molestias.

En cuanto a los contenidos del documento, son bastante diversos y completos, abarcando tanto aspectos técnicos de ingeniería del software como manuales de usuario, descripciones y capturas de pantalla del mismo, etc. En principio, y dado la extensión final del mismo, recomendamos a lectores que no dispongan de mucho tiempo la lectura de los capítulos I (introducción), III (que describe la motivación del proyecto y lo que trata de aportar), IX (características de la plataforma) y XIII (conclusiones). Con esto el lector puede hacerse una buena idea de cómo funciona la aplicación sin conocer cómo está hecha por dentro. Además el grueso de dicha información se encuentra tanto en inglés como en español en el capítulo II (resumen).

Sobre la bibliografía y lista de referencias del final hay una en concreto cuya lectura recomendamos especialmente. Ésta es [1], “Una Aproximación Documental para la Creación de e Integración de Juegos Digitales en Entornos Virtuales de Enseñanza”, tesis doctoral escrita por Pablo Moreno Ger, en la que está inspirada la motivación investigadora de este proyecto.

Junto con la memoria y prototipos hemos desarrollado dos juegos de prueba que pueden ejecutarse para completar la formación sobre <e-Adventure3D>. La primera, una simple aventura de química que pone a prueba el valor educativo de los juegos es descrita en el capítulo XII (caso de estudio). El segundo es una demo técnica de unos 30 minutos de duración que muestra todo lo que se puede hacer con <e-Adventure3D>.

Por último, toda la información está disponible en http://e-adventure.e-ucm.es. Por favor, no dude en contactar con nosotros para dudas, sugerencias o informar de errores en: e-adventure@e-ucm.es.

4. About this document

Welcome to the <e-Adventure3D> platform and thank you to take your time to read this document. Before we start, we would like to comment some aspects regarding its structure and contents.

Firstly, the document is organized in chapters chronologically (the first chapters describe initial approaches to the project; the middle chapters describe its features and the development process and lastly tests and conclusions). Most of them have been written in English to promote the use of the platform in foreign organizations and colleges, with exception of some appendices and the case study which due to a lack of time were written in Spanish. Our sincere apologizes for the inconveniences.

Regarding the contents, those are quite wide and complete, covering both technical features of software engineering and user manual, descriptions and screenshots of the project, etc. A priori and considering the extension of those contents we recommend readers with no too much time to spare to read chapters I (introduction), III (which describes the motivation of the project), IX (features of the platform) and XIII (conclusions). With all these the reader will acquire a good understanding of what the platform is and what can be done with it without knowing how it was developed. Besides, the main parts of such recommended reading can be found in Spanish as well in chapter II (summary).

About the bibliography and reference list attached at the end of this document, there is one which we strongly recommend to read. That is [1], "A Documental Approach to the Creation and Integration of Digital Videogames in Virtual Learning Environments", the Ph.D. thesis written by Pablo Moreno-Ger, which is the inspiration of the project from a research point of view.

Along with this report and the applications developed we have created two test games which can be run to gain a deeper knowledge of <e-Adventure3D>. The first is a simple chemistry adventure which puts to the test the educational value of the games, and its results are described in chapter XII (case study). The second is a tech demonstration which lengths about 30 minutes and shows all what <e-Adventure3D> can do.

And at last but not least, all the information of the project is available online on http://e-adventure.e-ucm.es. We hope you find interesting the platform. Please, do not hesitate in get in contact for any doubt, suggestions and bug reporting at: e-adventure@e-ucm.es.

Capítulo II

Resumen

1. Introducción. Motivación y Objetivos

1.1. e-Learning y videojuegos educativos

Durante las últimas décadas la tecnología ha experimentado un crecimiento exponencial, especialmente en lo referente a las tecnologías informáticas. La velocidad de cómputo de los procesadores y sistemas actuales ha crecido de forma exponencial, pasando de los 66MHz del procesador Intel 80486DX2 (1992) a la última generación de procesadores multi núcleo, cuya velocidad de reloj (por núcleo) es actualmente del orden de GHz. Además Internet ha supuesto una revolución en múltiples áreas tales como los negocios online, el sector de las comunicaciones, tiempo de ocio y entretenimiento, etc.

En resumen, los avances tecnológicos de las últimas décadas (especialmente Internet) han afectado a casi todas las facetas de nuestra vida diaria. Eso incluye también el campo de la enseñanza, en el que las TIC se vienen aplicando desde la década de los 90 en lo que se ha llamado *e-Learning*. Y el progreso en este campo ha sido sorprendente. Los modernos sistemas de e-Learning han evolucionado de ser meros repositorios de contenido estático en los primeros años 90 a a los imensos y complejos sistemas de contenido interactivo que cubren todo el proceso de aprendizaje para todos los roles implicados. Los profesores disponen de herramientas para evaluar y guiar a los alumnus durante el proceso; los diseñadores de conteidos (papel desempeñado muchas veces por los propios profesores) pueden en estos sistemas crear, almacenar, gestionar y presentar contenido digital que se almacena en un repositorio central, etc. Estos modernos sistemas de aprendizaje se conocen en ingles como *Learning Management Systems* (LMS) o Sistemas de Administración del aprendizaje en español, que en la mayor parte de los casos se basan en entornos web ya que estos contenidos son fáciles de crear, almacenar y sobre todo, de distribuir a los estudiantes en todo tipo de situaciones y contextos, debido a lo extendido de Internet [4].

Captura del campus virtual de la Universidad Complutense de Madrid, un servidor de aprendizaje WebCT [40], que es uno de los más extendidos hoy en día.

Hay muchas ventajas de este tipo de aprendizaje basado en contenido web. Los alumnos se ven obligados a adoptar un papel más activo en el proceso de aprendizaje y a desarrollar más sus capacidades de auto aprendizaje y auto evaluación, en contraposición al proceso pasivo mediante el cual los profesores transmiten el conocimiento a los alumnos que sólo tienen que sentarse y escuchar. Además los instructores pueden utilizar estas herramientas para dinamizar los cursos y fomentar la colaboración entre estudiantes [16] (el uso de foros y otras herramientas de comunicación está muy extendido actualmente y prácticamente todos los LMS lo soportan).

Sin embargo, también hay algunas desventajas. La desconexión entre los integrantes del proceso de aprendizaje dificulta la supervisión del mismo por los instructores, lo que en algunos casos imposibilita que se cumplan los objetivos educativos. Este problema se acentúa para aquellos alumnos que, por el motivo que fuera, requieren un mayor seguimiento. Además los cursos online normalmente experimentan altas tasas de abandono debido a la frustración de los alumnos y a su falta de motivación [36, 37]. Una alternativa para solventar esta desconexión es lo que se conoce como *b-Learning* (blended learning o aprendizaje mezclado), en la cual se mezclan aproximaciones tradicionales con otras más modernas basadas en TIC. De todas maneras, éstos son problemas que requieren ser abordados más profundamente.

Por otra parte, los videojuegos han sido identificados como una alternativa interesante en educación debido a ciertas características que presentan y que los hacen útiles para el aprendizaje [3, 26, 28]. Pedagogos y sociólogos reconocen que

los mamíferos (especialmente los humanos) aprenden jugando (en otras palabras, que la forma más natural de aprender es jugando). Eso supone que una experiencia de aprendizaje satisfactoria debería ser entretenida para el estudiante (el entretenimiento es una consecuencia del buen aprendizaje). Además, los videojuegos se han convertido en la mayor industria del entretenimiento (ahora mismo la industria del videojuego factura más que el cine) y su aceptación no solo se limita a niños sino a todo tipo de gente en términos de edad, raza y condición social. En contraste con la falta de atención que muestran los alumnos (mito de los 10 minutos de atención) los videojuegos consiguen mantener a sus usuarios concentrados durante horas en la resolución de tareas complejas que normalmente implican un alto razonamiento y habilidad para la resolución de problemas.

Otra ventaja es que los videojuegos modernos producen una alta interacción del jugador con el mundo representado en el juego (es decir, el dominio de estudio), que es una buena fuente de información que puede usarse para guiar a los alumnos (¿Está dando vueltas? ¿Está perdido?), para evaluarlos (¿Cuántos intentos ha necesitado para completar una tarea? ¿Cuánto tiempo necesitó para resolver un determinado problema?), o para adaptar el juego según el perfil del estudiante (por ejemplo teniendo en cuenta el nivel de conocimiento inicial). Finalmente, los videojuegos se pueden usar para promover la colaboración y competición entre estudiantes.

Teniendo en cuenta todas estas ventajas, los videojuegos pareen ser una interesante innovación en e-Learning, un campo que ha sido criticado debido a la baja motivación que provoca en los alumnos [36]. Además la discusión ya no está en si los juegos pueden enseñar y aportar beneficios a la experiencia de aprendizaje, sino en si esto merece la pena el coste y el esfuerzo. La principal barrera para la introducción de videojuegos en contextos educativos es su alto coste de desarrollo (de acuerdo con Aldrich en [16] 15 personas-año de media para videojuegos educativos y entre 100,000$ y 500,000$ según Michael & Chen en [41]), que están varios órdenes de magnitud por encima del coste de aproximaciones más tradicionales.

Otra desventaja es que los videojuegos no pueden enseñar por sí mismos. Simplemente dar un barniz a los contenidos con un juego no hará que los estudiantes aprendan. Para que los videojuegos puedan aportar experiencias educativas deben ser desarrollados equilibrando el factor educativo y el factor de entretenimiento [19]. Además deben integrarse en el resto del currículum y el transcurso de los juegos monitorizado para que los instructores puedan comprobar que los objetivos educativos son alcanzados.

Así que en este momento la discusión está centrada en si la experiencia de aprendizaje mediante videojuegos educativos es lo suficientemente buena para que compense el coste de introducirlos en el sistema educativo, ya que no hay suficientes pruebas a su favor. Por otra parte, los problemas del alto coste de desarrollo, el soporte de monitorización del transcurso de los estudiantes en el juego y su introducción como un recurso más en el sistema educativo son temas que deben ser abordados. Ahí es donde el proyecto <e-Adventure3D> trata de contribuir. La plataforma <e-Adventure3D> tratará de aportar un entorno complete de autoría para el desarrollo de videojuegos educativos de coste reducido, que puedan ser introducidos en entornos online y que soporten la motorización de los alumnos en los juegos.

1.2. Estudio del dominio: el desafío de las tres dimensiones.

En el párrafo anterior se han descrito las ventajas que los videojuegos pueden aportar a la educación. En esta sección llevamos a cabo un breve estudio del dominio, incluyendo un análisis de las experiencias con videojuegos en contextos educativos (sección 1.2.1) y las herramientas de autoría que podemos encontrar para la creación de videojuegos (sección 1.2.2) como estudio de las aplicaciones parecidas (trabajo relacionado) que pueden encontrarse en el mercado.

1.2.1. Experiencias con videojuegos en educación

Los videojuegos ya han sido utilizados en contextos educativos gracias a diversas iniciativas. La alternativa más sencilla (pero sin embargo efectiva), consiste en utilizar "juegos directamente sacados de la estantería" (Off-The-Self games, juegos desarrollados con propósitos comerciales que han sido directamente cogidos del mercado). Algunos de estos juegos son suficientemente ricos en contenido como para servir como material de aprendizaje.

Este es el caso del juego de estrategia *Civilization*, uno de los juegos más vendidos de la historia. El realismo de los escenarios históricos que aparecen en el juego ha sido aprovechado como material complementario en clases de historia (aunque los hechos históricos que tienen lugar en el juego pueden ser inexactos el alumno adquiere una percepción precisa de cómo era la vida en tiempos pasados). En [20] se describe una experiencia con estudiantes de un curso de historia con Civilization III.

Imágenes de los juegos *Civilization III* y *Sim City*.

Otro buen ejemplo es el caso de la saga *SimCity*. En estos juegos los jugadores deben administrar todos los aspectos de una ciudad como si estuvieran en el papel del alcalde. El éxito del título facilitó la aparición de otros juegos como *SimFarm*, *SimHealth* o *SimEarth*, publicados por los estudios Maxis, y que han sido aplicados en contextos educativos tal y como se describe en [21, 22][1].

Por otra parte, otros videojuegos han sido especialmente diseñados para cubrir la enseñanza de un dominio concreto. Este es el caso de *The Monkey Wrench Conspiracy*, publicado por *Think3,* que instruye en el uso de la aplicación de diseño industrial creada por la propia compañía. Otro ejemplo es *Virtual Leader*, desarrollado por *SimuLearn* para la enseñanza de un concepto tan complejo como es el "liderazgo". Por último podemos destacar el juego *Objection!* (protesto), desarrollado por *TransMedia* y que ha sido certificado como una herramienta válida para obtener créditos en los estudios de derecho para el programa de *Continuing Legal Education* (Formación Legal Continuada - CLE) en EE.UU[3].

Imágenes de *Virtual Leader* (izquierda) y *Objection!* (derecha).

[1] Otras experiencias con videojuegos comerciales se describen en [1], de donde se han obtenido estos ejemplos.

Un caso que merece especial mención es *Second Life* ® (Segunda vida) [8], de *Linden Lab*. *Second Life* es un mundo virtual 3D creado dinámicamente por sus residentes (gente representada en el mundo de Second Life por un avatar que ellos mismos eligen y a través del cual interactúan con el mundo). En Second Life los residentes pueden llevar a cabo con sus avatares múltiples tareas de la vida real, tales como personalizar su apariencia (ir a la peluquería, cambiarse de ropa), producir contenido digital (cuyos derechos de autor pertenecen a su creador como en la vida real) o comprar o vender sus propias pertenencias (incluyendo las intelectuales) usando dólares Linden, que también pueden ser convertidos en dólares americanos reales. La interacción en este mundo es tan rica que cientos de organizaciones como IBM o partidos políticos han creado sus propios mundos virtuales en Second Life.

Imagen de una clase online en Second Life de la facultad de derecho de la Universidad de Harvard.

Motivados por la alta aceptación de la simulación Second Life muchas facultades, colegios y otras organizaciones educativas han decidido beneficiarse de sus posibilidades educativas [7, 9] creando su propio campus virtual donde los alumnos pueden asistir a clases magistrales, participar en eventos con otros alumnos y profesores (promoviendo de esta manera el aprendizaje colaborativo) o acceder a recursos educativos online. Como se describe en [5, 6], Second Life es una buena herramienta para experimentar sobre diversos temas sociológicos con alumnos, para el entrenamiento en temas comerciales, o en general para dar clases magistrales de todo tipo de asignaturas.

Aunque todas estas experiencias han obtenido resultados satisfactorios, la aplicabilidad de este tipo de juegos está muy limitada. Éstos son productos cerrados, cuya ausencia de flexibilidad entorpece la reusabilidad de los contenidos en otras circunstancias y obliga al instructor a utilizar tests antes y después para comprobar que la experiencia educativa está siendo efectiva [39].

Además, los instructores no pueden intervenir directamente en el desarrollo de los juegos; de esta manera el diseño instruccional de los mismos se resiente (o incluso en algunos casos se ve relegado a un segundo plano), ya quienes tienen los conocimientos adecuados para diseñarlos no puede tomar parte en la implementación de los juegos. Y lo que es más, sus altos costes de desarrollo están fuera del alcance de la mayoría de presupuestos destinados educación. Como resultado estas alternativas no parecen ser las más adecuadas para la educación ya que no merece la pena afrontar el gasto si los contenidos no se pueden reutilizar en diversos cursos (sobre todo cuando los contenidos educativos varían con frecuencia y por lo tanto los instructores deberían poder modificarlos).

1.2.2. Herramientas de autoría para la creación de videojuegos comerciales y educativos

Como la sección anterior expresa, los videojuegos son como “cajas negras”, productos cuya dificultad de mantenimiento y adaptación cuando su producción termina, y cuyos altos costes de desarrollo son prohibitivos para presupuestos educativos. Esto es un inconveniente ya que los contenidos educativos pueden variar (los currículos oficiales pueden variar, los contenidos pueden tener que ser adaptados para distintos cursos o niveles, etc.). Además los instructores necesitan poder jugar un papel activo en el proceso de desarrollo de los juegos educativos si no se quiere perder el valor educativo de los mismos (ni programadores ni desarrolladores de juegos sabrán que es lo que deben aprender los alumnos).

Algunas de estas desventajas pueden solucionarse (o al menos paliarse) mediante el uso de herramientas de autoría para el desarrollo de los juegos. Algunas son las iniciativas que han causado impacto de diferente consideración. Sin embargo, no hay herramientas de autoría para videojuegos que cubran todas las innumerables características que puede tener un juego moderno. Como consecuencia, aquellas iniciativas que han triunfado se lo deben a haber conseguido equilibrar la potencia expresiva de la herramienta y la simplicidad de la misma. Normalmente esto se consigue restringiendo el campo de acción a un único género de juego (first person shooters, juegos de deportes, juegos de estrategia, de plataformas, etc.). Éste es el caso de algunas aplicaciones que van

desde algunas plataformas de autoría simples y gratuitas hasta otras profesionales (y de pago).

Entre las más relevantes encontramos *The FPS Creator* [11] (utilidad para la creación de juegos de disparo en primera persona), *The 3D Game Maker* [14] y *Game Maker* [29]. Los dos primeros ejemplos permiten a los usuarios crear complejos juegos en 3D con tan solo unos clics de ratón, mientras que la última ha sido utilizada como herramienta de desarrollo en diversos proyectos de investigación académica, tal y como se describe en [23, 24, y 25]. Otras iniciativas son *Alice* [30] y *Mission Maker* [31], herramientas que están especialmente pensadas para aplicaciones educativas.

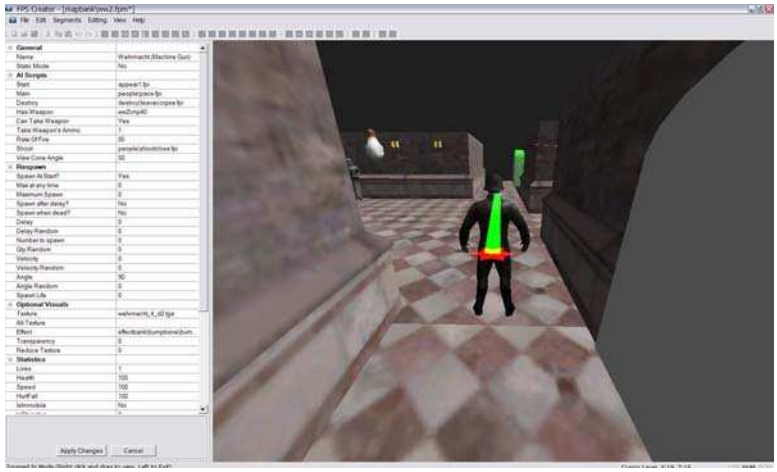


Imagen de la herramienta FPS Creator.

Sin embargo, no todos los juegos presentan las mismas características para su aplicación educativa. Como se describe en [34], casi todos los géneros tienen rasgos que pueden utilizarse para enseñar algo, pero el más adecuado según algunos autores [18, 26, 27, 28] es el género de aventuras. Cuando nos referimos a este género hablamos de juegos como *Myst*™ o *Monkey Island*™, en los que el jugador debe completar tareas mediante la exploración de su entorno y la interacción con objetos y otros personajes. La siguiente sección describe de forma más precisa las características que hacen que este tipo de juegos sea especialmente adecuado para la educación, pero por ahora simplemente destacar que en esos juegos lo importante es la narración (esto es los contenidos que han de ser transmitidos).

Imágenes de diferentes juegos de aventuras de Lucas Arts. Las dos primeras corresponden a títulos clásicos de dos dimensiones tales como *The Curse of Monkey Island* ® y *Full Throttle* ®. Los dos últimos muestran sus últimas aventuras, que fueron ya desarrolladas en tres dimensiones (*Escape from Monkey Island* ® y *Grim Fandango* ®).

Existen algunas herramientas para crear juegos de aventuras que permiten a autores sin conocimientos de programación crear sus propias aventuras. Entre ellas *Adventure Game Studio* [32] y *Adventure Maker* [33] son quizás las más populares.

Especial mención merece una plataforma de reciente creación: el proyecto <e-Adventure> [2] (<e-Adventure2D>), cuya última versión publicada (0.4b) sigue siendo una beta, pero cuya efectividad para la creación de aventuras educativas ha sido probada en varias aplicaciones [45, 46, 47]. Quizás esta sea el único ejemplo de una herramienta de autoría para juegos de aventura que ha sido especialmente diseñada para su uso por instructores y con fines educativos.

Aunque se ha probado la eficacia de estas herramientas para la creación de juegos educativos, los juegos producidos están limitados a representaciones 2D que se alejan de las tendencias actuales de los modernos juegos de aventura comerciales, que normalmente se crean en entornos 3D (como por ejemplo *Escape from Monkey Island* o *Grim Fandango*). La herramienta The *3D Adventure Studio* [12] es prometedora en ese aspecto, pero su desarrollo parece parado en

este momento.

1.3. Objetivos generales

Como se ha expuesto en las secciones anteriores, los videojuegos presentan características muy interesantes que pueden ser utilizadas para enseñar. La discusión ya no está en si los juegos pueden enseñar [3], sino en si merece la pena el esfuerzo. Los casos de estudio en los que se han aplicado videojuegos en educación siguen siendo escasos, aunque algunos hayan obtenido buenos resultados. Además su coste es excesivo para estas aplicaciones, y la creación y mantenimiento de los contenidos por instructores, quienes tienen la responsabilidad de conducir la experiencia de aprendizaje, son procesos complicados cuando no hay herramientas de autoría de por medio. Y lo que es más, la eficacia de la experiencia de aprendizaje no puede ser verificada ya que no existen mecanismos para ello en los juegos convencionales.

Una alternativa para abordar estos problemas (la cual supone la base de este documento) consiste en desarrollar herramientas de autoría para el desarrollo de videojuegos educativos sin necesidad de conocimientos de programación. De esta manera, los instructores (quienes raramente tienen ese conocimiento) pueden intervenir directamente en la producción de los juegos, y más tarde acceder a los contenidos y modificarlos. Además, la encapsulación de las tareas de programación es la clave para reducir los costes de desarrollo (el coste total de desarrollo prácticamente se limita al coste de producir los recursos artísticos).

Además, el género más adecuado para aplicaciones educativas parece ser el de aventuras. Hay herramientas de autoría para disponibles para crear estos juegos, pero los ejemplos más relevantes producen juegos en 2D, que no van acorde a las tendencias actuales. Es cierto que los videojuegos educativos no necesitan utilizar las últimas tecnologías ya que persiguen objetivos distintos que la industria de entretenimiento, pero es muy aconsejable que estos se parezcan lo más posible a lo que los alumnos esperan encontrar en un videojuego. Si las aventuras gráficas actuales son en tres dimensiones (lo que viene a ser una consecuencia de las demandas del público) es lo que un alumno esperará en estos juegos. Además el realismo y posibilidades de interacción en los juegos en 3D son mucho más ricos que en mundos en dos dimensiones ya que la exploración en el primer caso es más realista (hay cosas que no pueden representarse con precisión en un mundo en dos dimensiones, o que al menos pueden representarse mejor en un mundo en tres dimensiones).

De los párrafos anteriores surge el primer gran objetivo del proyecto:

(1) *Producir un entorno de autoría para la creación y ejecución juegos de aventura educativos en tres dimensiones que no requieran conocimientos de programación ni de desarrollo de videojuegos para su producción, a un coste bajo.*

Como se ha expuesto, hay algunas buenas herramientas de autoría para el desarrollo de videojuegos, pero éstas se centran en un desarrollo comercial tradicional que difiere de juegos educativos, de los que se espera tengan, características específicamente educativas. Entre ellas las más destacadas son la capacidad de auto evaluar al alumno (assessment), de adaptar los juegos según el perfil de los alumnos (adaptation), y la capacidad de integrar los juegos en los actuales Sistemas de Administración del Aprendizaje (LMS) de tal forma que los resultados puedan adjuntarse al perfil de los alumnos.

Como consecuencia, éste es el segundo objetivo del proyecto:

(2) *Dotar a la plataforma a desarrollar con mecanismos para la auto evaluación y adaptación de la experiencia de aprendizaje. La plataforma debe poderse comunicar con un LMS para que la evaluación se pueda guardar en el perfil del alumno y se pueda llevar a cabo la adaptación de acuerdo al mismo.*

Como ya se ha mencionado, el propósito general del proyecto es reducir los altos costes de desarrollo de los juegos educativos mediante prácticamente la eliminación de los costes de programación e implementación de los juegos. No es la motivación de este proyecto tratar de reducir los costes de la producción de los recursos artísticos, ya que poco podemos contribuir en esa faceta. Hay multitud de herramientas para la creación de imágenes, sonidos, videos o modelos 3D, y difícilmente seríamos capaces de mejorarlas o simplificarlas. Eso es una tarea que corresponde a otras personas más cercanas al mundo del diseño artístico.

Finalmente, hay un objetivo subyacente que nos gustaría señalar. Las experiencias con herramientas para la producción de aventuras en dos dimensiones han obtenido buenos resultados. Limitar el ámbito de estas herramientas a ese género permite extrapolar los elementos comunes de las mismas y simplificar su edición. Por otra parte, la producción de juegos en dos

dimensiones es razonable desde un punto de vista técnico ya que éstos se componen únicamente de imágenes que se renderizan una sobre la otra en el sistema de visualización. Sin embargo, no hay indicios a priori de que el mismo razonamiento sea válido para juegos en tres dimensiones. En este caso su complejidad técnica escapa a nuestro control. Además la alta capacidad expresiva de los juegos en tres dimensiones hace que el proceso de extrapolar los elementos comunes sea mucho más complejo ya que debe equilibrarse el poder expresivo de la plataforma y su sencillez de forma muy cuidadosa. De otra forma, un exceso de expresividad podría abrumar a instructores sin mucha habilidad con los ordenadores, mientras que una aplicación demasiado sencilla produciría juegos demasiado parecidos y poco vistosos, dejando escapar el factor de motivación que se persigue en los juegos. Encontrar dicho equilibrio será una tarea de diseño clave para llegar a cumplir los objetivos propuestos.

1.4. Descripción del trabajo

La sección anterior establece cuales son los principales objetivos de nuestro proyecto. En la presente daremos una breve descripción de cómo pensamos desarrollar la aplicación para alcanzar los objetivos propuestos.

Como el objetivo (1) describe, vamos a producir una herramienta para la creación de aventuras gráficas en 3D con fines educativos. A priori, vamos a seguir procedimientos de desarrollo similares a los seguidos en <e-Adventure2D> para beneficiarnos de la experiencia de la que goza el departamento gracias a su desarrollo. La plataforma <e-Adventure3D> estará compuesta de esta manera por dos aplicaciones: un motor de juegos que será el encargado de ejecutar los juegos, y una herramienta de edición para ayudar a crearlos.

La parte principal del proyecto es, a priori, el desarrollo del motor junto con el lenguaje necesario para implementar los juegos. Siguiendo las ideas descritas en [1], tomaremos una aproximación documental para representar nuestros juegos. De esta forma los juegos se escriben en documentos de texto simple, que son “legibles por humanos” y que el motor se encargará de interpretar y ejecutar en combinación con los recursos artísticos del juego, que son dejados a parte (se separa el guión del juego de su representación). De esta manera los instructores pueden escribir guiones ejecutables para los juegos con la ayuda de un editor de textos, lo que reduce los costes de desarrollo tanto de los juegos como del propio editor.

Para ello necesitamos diseñar un lenguaje para la descripción de los juegos (lenguaje del guión), basado en tecnologías XML, y que logre un equilibrio

adecuado entre poder expresivo (lo que puede ser expresado con el lenguaje) y simplicidad. Por una parte, si el lenguaje no es suficientemente expresivo los juegos producidos serán demasiado similares unos de otros y difícilmente consigan enseñar algo por falta de recursos. Por otra parte, si el lenguaje es demasiado complejo los instructores verían complicado desarrollar sus propios juegos utilizando la plataforma.

Sin embargo, es probable que los instructores encuentren escribir todo el guión de un juego en 3D en XML directamente una tarea demasiado ardua (hay que tener en cuenta que las sintaxis XML son muy rígidas y a veces es pesado escribir estos documentos a mano). Además, aunque este género de juegos no cubre problemas complejos como el diseño de la inteligencia artificial o complejos sistemas de partículas, posiblemente un instructor necesite ayuda para colocar los personajes y objetos en las escenas (para ello se requiere un posicionamiento 3D, y los elementos pueden requerir rotaciones o escalamientos que no son normalmente tareas sencillas), o crear conversaciones complejas con cientos de líneas de diálogo. Para ayudar a llevar a cabo aquellas tareas demasiado complejas o que requieran demasiado tiempo vamos a desarrollar una herramienta de edición, como apoyo a la creación manual de los guiones.

2. Características de la plataforma

2.1. El motor <e-Adventure3d>

El motor < e-Adventure3d> es la unidad que procesa el archivo de tipo EA3D (que son archivos ZIP renombrados). Este archivo contiene todos los datos necesarios para interpretar y para ejecutar el juego. Por lo tanto el EA3D contiene lo siguiente:

- **Recursos artísticos**: modelos tridimensionales, imágenes, sonidos y videos utilizados en el juego.
- Los **documentos XML**: Estos documentos controlan el juego entero, dando indicaciones al motor sobre cómo proporcionar los recursos artísticos y sobre los elementos narrativos de la aventura. Hay cuatro tipos de documentos XML cuyas sintaxis del lenguaje se puede consultar en el capítulo de apéndices en forma de DTD. Esos archivos son:

  - **Archivos con el guión del juego**. El flujo narrativo de los juegos se estructura en capítulos, que son pequeñas partes del juego independientes unas de otras, fáciles producir y mantener y también útiles para no tener

que cargar en memoria todo el juego. La narración de cada capítulo esta descrita en un archivo XML diferente.

- **El descriptor de la aventura**, que contiene la información de carácter general sobre la aventura como el título, la descripción, los ajustes del GUI y los datos de la configuración para cada capítulo (qué archivos de reglas de evaluación automática y adaptación utilizar).

- **Perfiles de evaluación automática**. Cada capítulo puede estar unido con un perfil de 'assessment' para producir una evaluación automática del estudiante. Para llevar a cabo la evaluación, se definen reglas de 'assessment' en un documento XML.

- **Perfiles de adaptación.** Cada capítulo también se puede unir con un perfil de la adaptación. Esto se conoce como 'adaptation' y es la característica de los juegos que permite calibrar su comportamiento según el perfil del estudiante. Análogamente a la evaluación automática, la adaptación se define en términos de reglas que se almacenan en archivos XML.

Usando todos estos componentes el motor <e-Adventure3D> produce juegos ejecutables.

Esta parte de nuestro proyecto se ha desarrollado usando tecnologías Java (las tecnologías se detallan en el capítulo 'Design and implementation').

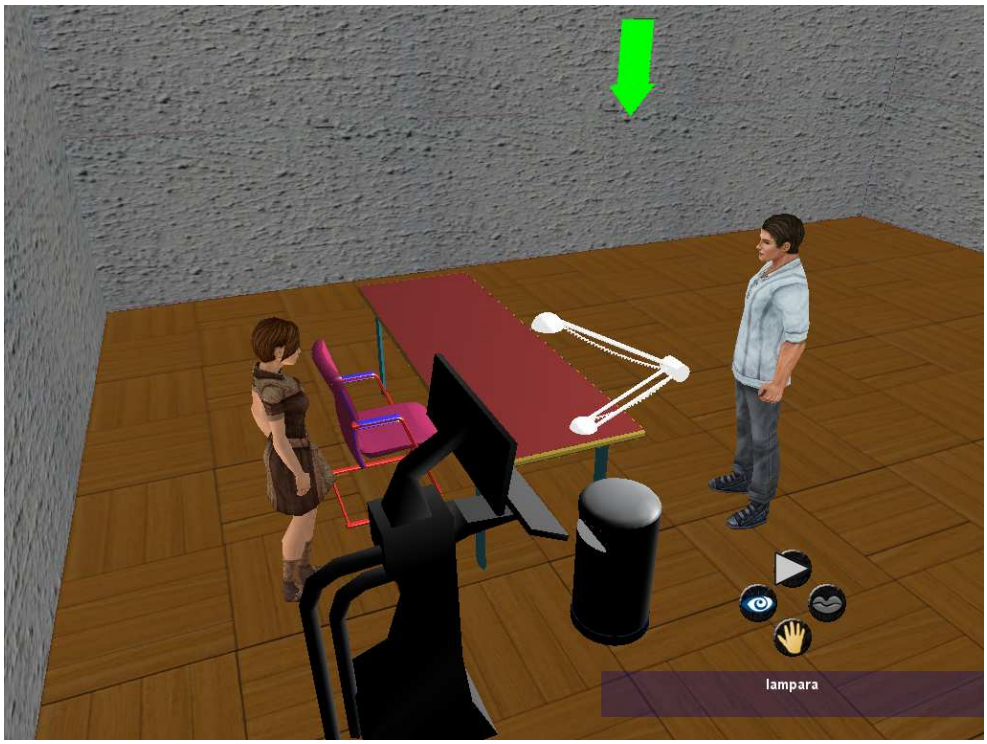
Pero, ¿cómo podemos conseguir un juego ejecutable a partir documentos XML y de recursos artísticos? Hay algunas características que las aventuras gráficas en tres dimensiones deben soportar como por ejemplo el movimiento del jugador, la información sobre los elementos cerca del jugador (que son mostrados por el HUD, del cual hablaremos más adelante), el inventario donde el jugador guarda los objetos que posee y las conversaciones entre los personajes. Una vez que tengamos la información de la aventura obtenida de los archivos de XML, mostraremos la información necesaria y los recursos artísticos en el momento conveniente usando JME. También permitiremos o denegaremos las acciones que el usuario puede realizar según la información que define el juego.

Vamos a ver las partes más importantes del motor:

El sistema de entrada recopila los datos sobre el estado de los dispositivos de entrada (el teclado o el mando) y los lleva al controlador adecuado (es decir, a los controladores del movimiento y de la interacción). Sin embargo, el problema es que cada mando es diferente, así que desarrollamos una utilidad de configuración de mandos para hacer estándar la identificación correcta de los botones. JME es

un motor 3D moderno desarrollado recientemente y aún no tiene todas las funciones que necesitamos, así que tuvimos que dedicar un gran esfuerzo a la mejora del sistema de entrada.

Una parte importante fue el diseño del **HUD**. El HUD se encarga de mostrar la información sobre los elementos accesibles por jugador y las acciones aplicables al objeto seleccionado en cada momento. Hay cuatro acciones posibles: usar, coger, examinar y hablar. Decidimos que estas acciones se podían asociar con los cuatro botones que generalmente tienen los mandos (y también dar esa estructura al HUD). Vamos a mostrar un ejemplo en el que el protagonista del juego está cerca de una lámpara. En la parte de abajo del HUD tenemos el panel informativo donde se escriben los nombres de los objetos accesibles (en este caso sólo la planta) y en el centro de ellos el nombre del elemento seleccionado. En el propio HUD resaltamos las acciones permitidas para el objeto seleccionado. En este caso particular podemos coger y examinar la lámpara así que se colorean los iconos correspondientes del HUD. También se pinta una flecha verde que apunta al objeto seleccionado para que no haya duda de cual es:



El inventario muestra los objetos que el jugador posee en el momento actual del juego. Para cada objeto, se muestra el icono que lo representa y su nombre. En ocasiones se permite que un objeto del inventario se utilice con otros elementos (del interior o del exterior del inventario).Si sucede esto, el HUD lo indica destacando el icono de la acción. De todos los artículos del inventario el que está

seleccionado en un momento dado es el que se encuentra en el centro:

El menú del juego se puede utilizar para cambiar algunos parámetros del juego. Permite cambiar las características de las conversaciones (habilitando o deshabilitando el sonido o los subtítulos). También se utiliza para guardar y cargar partidas. Como vemos en la siguiente imagen también se puede utilizar para abandonar el juego:

Entre capítulo y capítulo del juego, utilizamos **pantallas de carga** para indicar al jugador que el juego no está bloqueado. Una pantalla de carga tiene la siguiente apariencia:

2.2 La herramienta de edición <e-Adventure3d>

La meta principal del editor de juegos <e-Adventure3D> es hacer más fácil la manera de escribir el guión de los juegos (en general, crear el archivo completo EA3D) que serán ejecutados en el motor <e-Adventure3D>. El redactor se convierte en una necesidad desde el momento en que los usuarios intentan crear un juego que implique unas pocas escenas.

Como el motor, el editor fue desarrollado usando tecnologías Java.

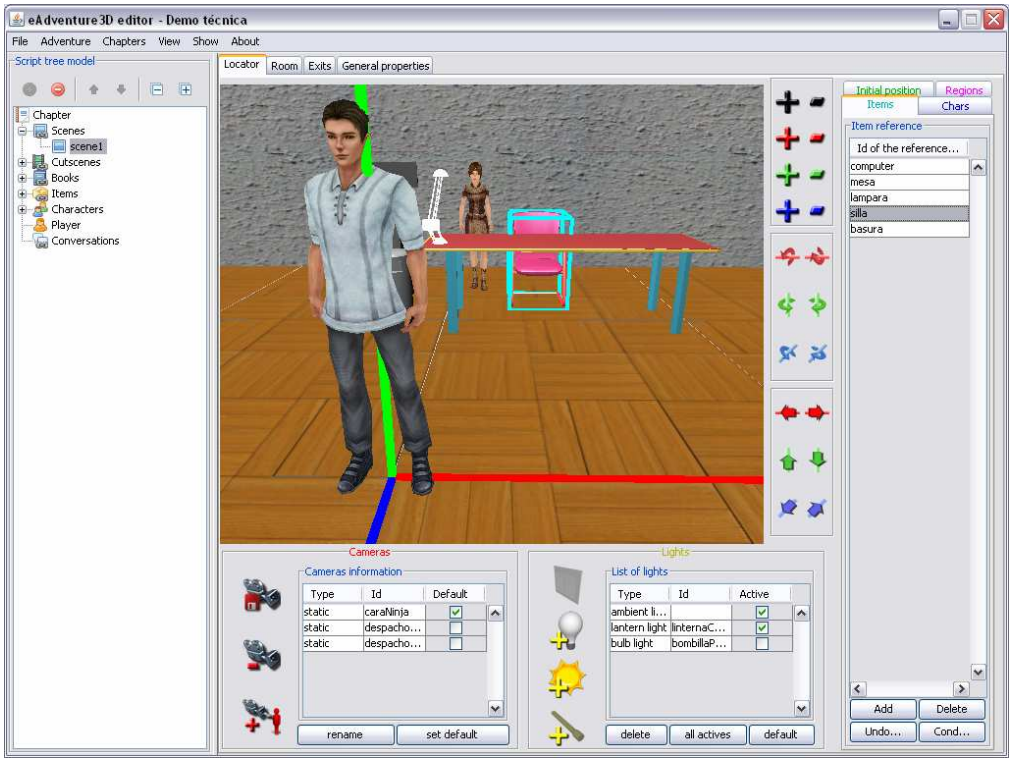
Hay una compatibilidad completa entre el editor y el lenguaje <e-Adventure3D> (y por tanto con el motor); por ello, todo lo que se puede escribir en los documentos XML se puede redactar gráficamente con el editor. Entonces, si se puede tener un juego usando cualquiera de las dos maneras, ¿por qué hemos desarrollado un editor? La respuesta a esta pregunta se volvió obvia cuando intentamos crear nuestro primer juego de prueba escribiendo los guiones. Aunque tenía solamente un capítulo con tres escenas, veinte objetos y siete personajes; cada uno de los personajes tenía algunas conversaciones y ambos, personajes y objetos tenían transformaciones[2] asociadas para introducirlos en las escenas. El hecho es que el archivo XML con el guión de este capítulo alcanzaba unas mil

[2] *Las transformaciones son operaciones realizadas a un elemento de una escena para escalarlo, rotarlo o trasladarlo.*

quinientas líneas (escritas a mano) que implicaron que dos personas dedicaran unos 10 días a desarrollar la aventura. El editor <e-Adventure3D> soluciona este problema. Permite que los fabricantes del juego creen las aventuras de una manera gráfica; así que no tienen que saber nada sobre XML o, en particular, sobre el lenguaje <e-Adventure3D>. Esto reduce notablemente la cantidad de horas del trabajo como luego comprobamos al desarrollar otro juego de prueba (la demo técnica). Este segundo juego de prueba tiene cuatro capítulos, con un total de más de dos mil quinientas líneas escritas de lenguaje <e-Adventure3D> y fue desarrollado en una semana y por una sola persona gracias al editor.

Consecuentemente el editor de juegos <e-Adventure3D> es una gran ventaja frente a escribir del documento XML, incluso aunque el usuario domine el lenguaje que hemos definido. Hay varios puntos fuertes que hacen que el editor sea una herramienta muy poderosa:

- Ventanas de previsualización utilizando el motor JME: Estas ventanas permiten que los usuarios previsualicen la aventura que están diseñando tal y como quedará en el motor cuando se ejecute. Son muy útiles de muchas maneras. Por ejemplo, una escena completa de la aventura puede ser previsualizada y los usuarios podrán añadir las instancias de objetos y personajes que quieran haciendo transformaciones en sus modelos tridimensionales para colocarlos donde se desee y con el tamaño que se desee sin necesidad de ejecutar el motor para comprobar que hemos acertado con estas transformaciones (como pasaría si tuviesen que escribirlo). En la imagen siguiente podemos ver la previsualización de una escena de la demo técnica. Aparece seleccionada una silla, a la cual podríamos transformar para situarla donde queramos (el manual de usuario muestra la información detallada sobre este proceso).

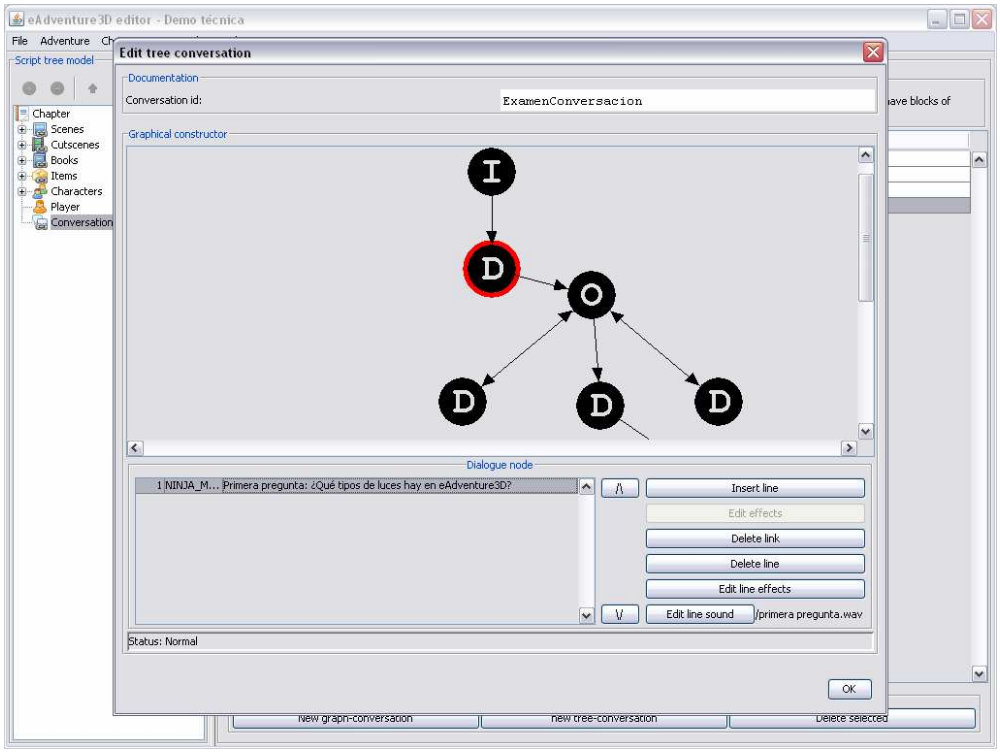


En las pantallas de previsualización, los usuarios también pueden mover la cámara donde quieran (para ver la escena desde donde convenga en cada momento) con tan solo arrastrar y soltar el ratón sobre dichas ventanas hasta el lugar donde deseen.

- Fácil manejo de cámaras: En la sección siguiente, se explica con más detalle que los usuarios deben tener conocimientos de geometría básica para definir cámaras fotográficas directamente en el lenguaje <e-Adventure3D>. Esto no es un requisito cuando los usuarios crean juegos con el editor puesto que pueden crear ambos tipos de cámaras sin ninguna dificultad. Como hemos escrito antes, los usuarios pueden ver una escena de antemano completa y con un movimiento simple del ratón pueden colocar la cámara estática dondequiera que deseen y guardar la posición de la cámara utilizada para aplicarla posteriormente durante la partida que se juegue en el motor. También de manera visual se pueden crear cámaras en tercera persona. (Los detalles de cómo crearlas aparecen en el manual de usuario del editor).
- Control de identificadores: Otra dificultad de escribir una aventura era recordar todos los identificadores y donde las hemos utilizado. Los identificadores se utilizan para identificar muchas cosas tales como las conversaciones, personajes, objetos, escenas, cámaras, luces o 'flags'. No se pueden repetir identificadores, por tanto los usuarios deben recordar el nombre de objetos y de personajes o buscarlos en el documento XML

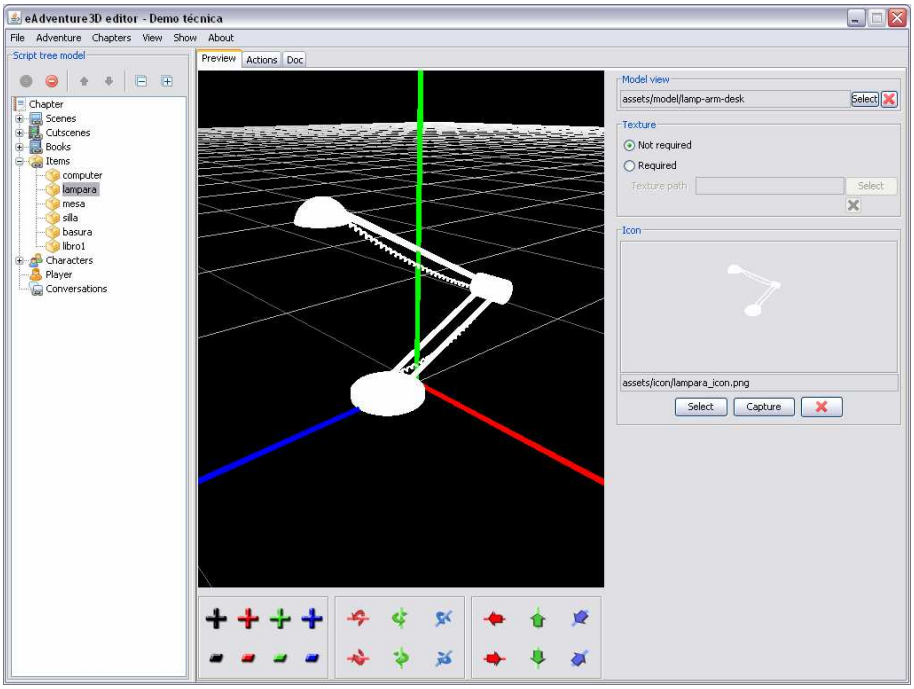
cuando no recuerden el nombre y tengan que referenciarlos en una escena. También, en caso de que renombren, por ejemplo, un objeto, tendrán que cambiar su nombre cada vez que lo hayan utilizado en el capítulo. Esta clase de cosas requiere tiempo y conducen a equivocaciones. Sin embargo esto no pasa en el editor, donde estas cosas se tratan automáticamente gracias al control de identificadores: no permite que los usuarios repitan identificadores, suprime todas las referencias a un elemento si este se suprime y permite que los usuarios renombren un elemento renombrándose automáticamente todas las referencias que existan.

- Edición gráfica de conversaciones: las conversaciones se representan de manera gráfica con grafos o con árboles (dependiendo de si la conversación puede tener ciclos o no). La imagen siguiente muestra el editor gráfico para conversaciones que permite que los usuarios escriban de manera ordenada una conversación entre varios personajes:



Para ver una descripción completa de las conversaciones y cómo crearlas por favor visite el apéndice A (manual de usuario). Por ahora basta con explicar que la representación en nodos permite que la conversación tome diversas trayectorias dependiendo de las elecciones tomadas por el jugador durante una conversación. Por lo tanto los nodos se distribuyen principalmente en nodos de diálogo (que contienen las líneas del diálogo) y nodos de opción (que contienen varias posibilidades para elegir).

- Creación automática de iconos: Algunos objetos de la aventura pueden ser almacenados en el inventario del jugador. Estos objetos deben tener un icono de dos dimensiones que los represente como ya vimos en la sección anterior hablando del inventario. El editor permite crear un icono en dos dimensiones a partir de la previsualización de un modelo tridimensional con tan sólo pulsar el botón 'Capture' como vemos a continuación:



Los creadores de juegos no tendrán que dibujar imágenes de cada objeto que vaya a ser guardado en el inventario del jugador gracias a esta característica. Esto reduce los costes temporales (suponiendo que el usuario tuviera que dibujar el icono) o económicos (en caso de que hubiera que comprar una imagen) de desarrollar un juego.

- Los recursos y archivos XML se almacenan automáticamente: Cuando hacemos referencia a un recurso nuevo mientras creamos una aventura con el editor de juegos, este activo se copia automáticamente en el archivo EA3D. La ventaja de esto es que los desarrolladores de juegos no tendrán que recordar agregar los recursos referidos manualmente. Por otra parte, la gestión de capítulos en una misma aventura también se puede hacer con el editor y, al igual que ocurre con los recursos, cuando añadimos nuevos capítulos o los modificamos, el archivo XML se agrega al archivo EA3D al guardar la aventura. Ocurre lo mismo con el descriptor y, en caso de que existan, también con los ficheros de reglas de adaptación y evaluación automática.

Como conclusión pensamos que nuestro proyecto no estaría completo sin el editor porque, como hemos visto en esta sección, éste marca la diferencia: permite que los usuarios definan aventuras sólidas completas en menos tiempo que si escribiesen el guión a mano y, además, juegos educativos complicados pueden ser creados por los usuarios que no estén familiarizados con el lenguaje <e-Adventure3d> (el cual el editor hace transparente al usuario) o, en general, con XML y tampoco es necesario que tengan ningún conocimiento de geometría.

2.3. Principales características de <e-Adventure3D>

Las DTD definen la estructura de un documento de XML, así que definimos nuestra propia DTD para el lenguaje <e-Adventure3D>. La idea era mantener la simplicidad del lenguaje escrito para la versión anterior del proyecto (<e-Adventure2D>) tanto como fuera posible. Sin embargo, en el caso de un espacio tridimensional la única diferencia no es sólo que ahora necesitamos una nueva coordenada (z) para representar un punto en el espacio. Por supuesto, las cosas no son tan simples. Hay características en <e-Adventure3D> que hacen que mantener la simplicidad del lenguaje se convierta en algo realmente complejo (por ejemplo el uso de cámaras). En nuestra opinión hemos conseguido simplificar el lenguaje tanto como es posible sin dejar a un lado el potencial expresivo de un espacio tridimensional. Además, por si aún así no fuese suficiente, tenemos el editor que hace que el lenguaje <e-Adventure3D> se convierta en algo transparente para los usuarios. Además de mantener la simplicidad, otro requisito era que el lenguaje fuera independiente del motor 3D de Java (JME), de modo que en un futuro el motor se pudiera sustituir para otro sin tener que modificar el lenguaje que hemos desarrollado.

En esta sección vamos a resaltar las características más importantes de <e-Adventure3D>. Para ello, la mejor manera es ir explicando el lenguaje <e-Adventure3D> ya que abarca todas las características que pueden tener nuestros juegos educacionales. Con el fin de explicar el lenguaje, vamos a ver cada una de las partes de la DTD que lo definen:

```
<!--E-ADVENTURE3D LANGUAGE-->
<!ELEMENT eAdventure3d ((scene | %cutscene;)+, book*, object*, player, character*, (%conversation;)*)>
```

Detallaremos cada una de estas partes y mostraremos algunos ejemplos del lenguaje escritos en XML junto con la representación gráfica de ese fragmento del lenguaje en el editor y/o en el motor, según sea más ilustrativo.

2.3.1 Escenas

En primer lugar vamos a explicar las **escenas**. Esta es la parte de la DTD que las define:

```
<!--SCENE-->
<!ELEMENT scene (documentation?, resources?,name, environment?, default-initial-position, regions?, exits?, objects?, characters?, view?)>
```

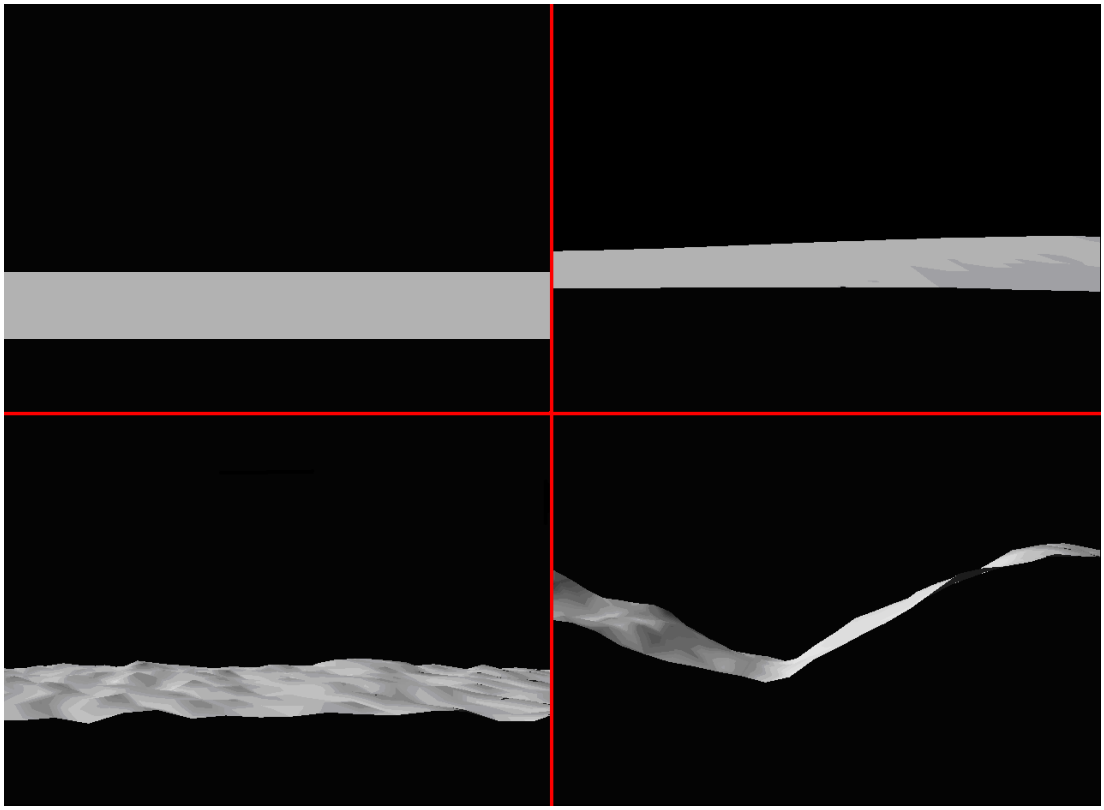
Vamos a explicar cada componente de las escenas.

Una de las tareas más simples de <e-Adventure2D> era crear el **entorno** ('environment' en la DTD) de la escena porque era sólo establecer una imagen de fondo. Un espacio tridimensional transforma la creación del entorno en algo más complicado. Nuestra meta era simplificar esta tarea a los usuarios, así que (como se puede observar en la definición de ambiente en la DTD) tenemos dos tipos de entornos predefinidos: *los espacios abiertos* y *los cuartos*.

```
<!--SCENE ENVIRONMENT-->
<!ELEMENT environment (openEnvironment | room)>
```

Los **espacios abiertos** son entornos definidos por un terreno rodeado por un cielo.

El terreno tiene como parámetros el tamaño, la textura y uno de estos aspectos dependiendo de la aspereza seleccionada por el usuario; puede ser: llano, liso, áspero o montañoso. A continuación vemos una muestra de estos cuatro tipos de terrenos:



Hay también otro tipo de espacio abierto que genera un terreno llano con una textura de agua. Los resultados son absolutamente satisfactorios ya que este terreno da la sensación de ser un mar con olas, sin que el usuario tenga que indicar nada a parte de establecer que el tipo de terreno es de agua. Desafortunadamente todavía está en una etapa experimental, así que no está garantizado que funcione en todos los sistemas.

En cuanto al cielo (en los juegos en tres dimensiones se conoce como 'skybox') se define a partir de 6 imágenes que forman un cubo que cubre el espacio tridimensional que engloba la escena.

Veamos la DTD completa de los entornos para luego poner algunos ejemplos:

```
<!ELEMENT openEnvironment (sky, terrain)>
<!ATTLIST openEnvironment
   fog-enabled (yes | no) "no"
>
<!ELEMENT terrain (terrainTexture?)>
<!ATTLIST terrain
aspect (plain|smooth|rough|mountainous | water) "plain"
width NMTOKEN #REQUIRED
depth NMTOKEN #REQUIRED
uri CDATA #IMPLIED
>
<!ELEMENT terrainTexture (textureScale?)>
<!ATTLIST terrainTexture
%uri;
>
<!ELEMENT sky (north, south, west, east, top, bottom)>
<!ELEMENT north EMPTY>
<!ATTLIST north
   %uri;
>
<!ELEMENT south EMPTY>
<!ATTLIST south
   %uri;
>
<!ELEMENT west EMPTY>
<!ATTLIST west
   %uri;
>
<!ELEMENT east EMPTY>
<!ATTLIST east
   %uri;
>
<!ELEMENT top EMPTY>
<!ATTLIST top
   %uri;
>
<!ELEMENT bottom EMPTY>
<!ATTLIST bottom
   %uri;
>
```

Veamos un terreno áspero con una textura de nieve y bordeado por un cielo de un paraje montañoso. En primer lugar, vamos a escribir el código XML y después a ver cómo se generará:

```
<environment>
   <openEnvironment fog-enabled="yes">
      <sky>
         <north uri="assets/sky/alpes_north.jpg"/>
         <south uri="assets/sky/alpes_south.jpg"/>
         <west uri="assets/sky/alpes_west.jpg"/>
         <east uri="assets/sky/alpes_east.jpg"/>
         <top uri="assets/sky/alpes_up.jpg"/>
         <bottom uri="assets/sky/alpes_down.jpg"/>
      </sky>
      <terrain aspect="rough" depth="300.0" uri="assets/terrain/scene0.jme" width="300.0">
         <terrainTexture uri="assets/texture/b19nature_landscapes391.jpg">
            <textureScale x="1.0" y="1.0" z="1.0"/>
         </terrainTexture>
      </terrain>
   </openEnvironment>
</environment>
```

Podemos definir fácilmente otro espacio abierto totalmente diferente. Por ejemplo, si elegimos uno llano con un cielo que forme el horizonte de una moderna ciudad y utilizamos una textura de aspecto de hierba, entonces obtenemos la imagen de un parque en medio de unos rascacielos. Este es el resultado visto en el motor:

El otro tipo de entornos predefinido lo denominamos cuarto. Hay dos tipos: los cuartos cerrados y los cuartos modelo.

```
<!ELEMENT room ((standardRoom | modelRoom), sky?)>
```

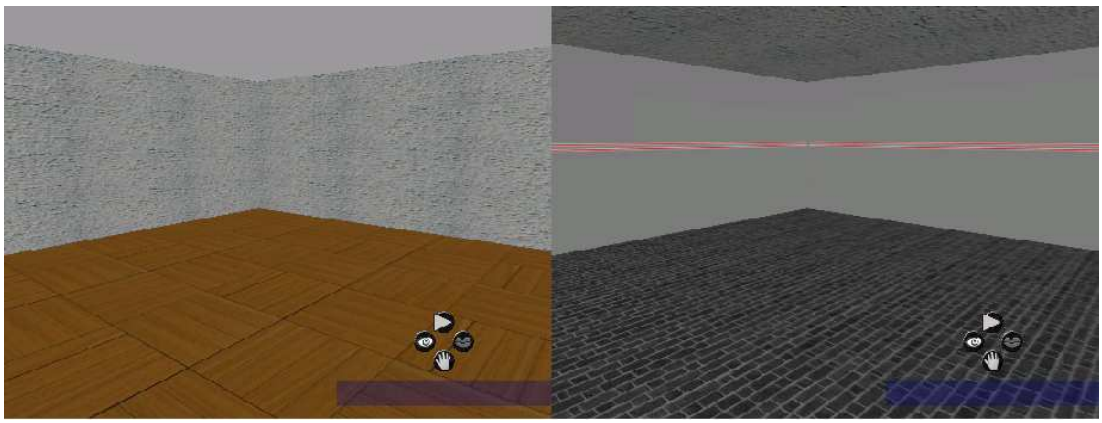
El **cuarto cerrado** es un cuarto simple con el tamaño que especifique el usuario. Se compone de cuatro paredes paralelas, de un techo y de un suelo con sus respectivas texturas. Los usuarios pueden cambiar las texturas fijadas por defecto por el editor con tan solo seleccionar la imagen deseada del sistema de ficheros. En este tipo de escena los usuarios pueden definir entre cero y cuatro salidas que se representan en la escena como puertas (una en cada pared). Los cuartos son muy útiles en las aventuras debido a su bajo coste de producción y cambian su aspecto totalmente con solamente cambiar las texturas.

La definición de un cuarto cerrado sigue esta parte de la DTD:

```
<!ELEMENT standardRoom (roomTextures)>
<!ATTLIST standardRoom
   width NMTOKEN #REQUIRED
   height NMTOKEN #REQUIRED
   depth NMTOKEN #REQUIRED
>
<!ELEMENT roomTextures (wallsTexture?, ceillingTexture?, floorTexture?)>
<!ELEMENT wallsTexture (textureScale?)>
<!ELEMENT ceillingTexture (textureScale?)>
<!ELEMENT floorTexture (textureScale?)>
<!ELEMENT textureScale EMPTY>
<!ATTLIST textureScale
%vector3;
>
<!ATTLIST wallsTexture
   %uri;
>
<!ATTLIST ceillingTexture
   %uri;
>
<!ATTLIST floorTexture
   %uri;
>
```

A continuación vamos a poner un ejemplo del cuarto cerrado. En primer lugar, se muestra el fragmento de XML que lo define. En segundo lugar, podemos ver dos veces el mismo cuarto, visto del mismo lugar pero con la salvedad de haber utilizado dos texturas distintas (cambiar la textura es cambiar sólo el atributo URI de la textura por el path de la nueva imagen).

```
<environment>
   <room>
      <standardRoom depth="300.0" height="100.0" width="300.0">
         <roomTextures>
            <wallsTexture uri="assets/texture/wall.JPG">
               <textureScale x="1.0" y="1.0" z="1.0"/>
            </wallsTexture>
            <ceillingTexture uri="assets/texture/ceilling.jpg">
               <textureScale x="4.0" y="4.0" z="4.0"/>
            </ceillingTexture>
            <floorTexture uri="assets/texture/floor.jpg">
               <textureScale x="4.0" y="4.0" z="4.0"/>
            </floorTexture>
         </roomTextures>
      </standardRoom>
   </room>
</environment>
```



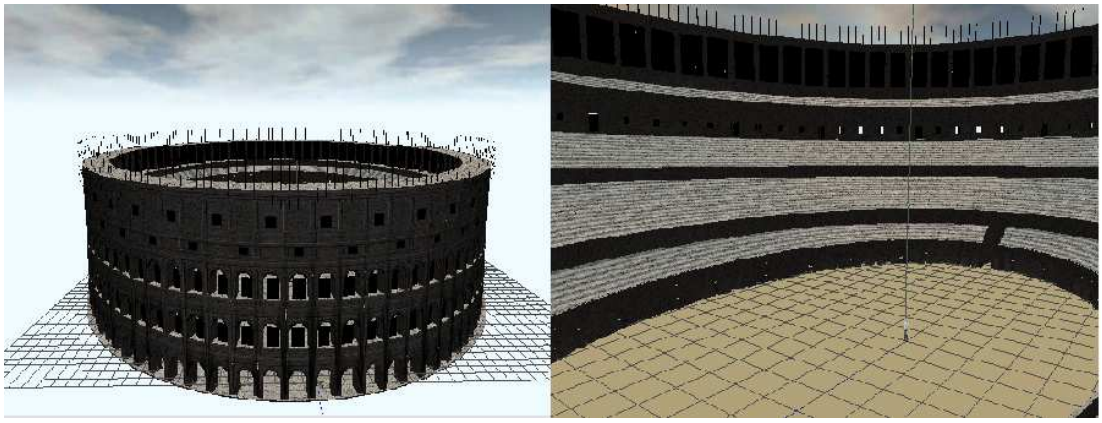
Los **cuartos modelo** son entornos definidos a partir de un modelo tridimensional. Elegimos el modelo que formará el cuarto, delimitamos la región por la que podrá moverse el jugador y escogemos (si se quiere) un cielo que lo bordee (al igual que en los espacios abiertos). A continuación vemos la definición en la DTD:

```
<!ELEMENT modelRoom (extension?,transformations?)>
<!ATTLIST modelRoom
   %uri;
   %positionR;
>

<!ELEMENT extension EMPTY>
<!ATTLIST extension
   x-min CDATA #REQUIRED
   z-min CDATA #REQUIRED
   x-max CDATA #REQUIRED
   z-max CDATA #REQUIRED
>
```

Por ejemplo, el modelo que define nuestro cuarto puede ser el coliseo de Roma. Este sería el código XML que lo define:

```
<environment>
   <room>
      <modelRoom uri="assets/model/COLI_L" x="297" y="103" z="5">
         <extension x-max="162" x-min="-250" z-max="180" z-min="-180"/>
         <transformations>
            <objectRotation>
               <rotation-axis-x angle="-90.0"/>
               <rotation-axis-y angle="0.0"/>
               <rotation-axis-z angle="0.0"/>
            </objectRotation>
            <objectScale>
               <scale-axis-x scale="1.975897"/>
               <scale-axis-y scale="1.975897"/>
               <scale-axis-z scale="1.975897"/>
            </objectScale>
         </transformations>
      </modelRoom>
      <sky>
         <north uri="assets/sky/sky_north.jpg"/>
         <south uri="assets/sky/sky_south.jpg"/>
         <west uri="assets/sky/sky_west.jpg"/>
         <east uri="assets/sky/sky_east.jpg"/>
         <top uri="assets/sky/sky_up.jpg"/>
         <bottom uri="assets/sky/sky_down.jpg"/>
      </sky>
   </room>
</environment>
```

En la siguiente imagen mostramos dos puntos de vista del entorno creado. Una visión general del coliseo y otra desde las gradas en la que se ve la arena:



Los cuartos modelo no tienen un coste tan reducido como los cuartos cerrados puesto que los usuarios tendrían que producir el modelo tridimensional (o contratar a un artista especializado en modelos tridimensionales). No obstante, la expresividad ganada introduciendo esta clase de entorno tiene mucho valor, ya que se pueden mostrar escenarios reales en los juegos sin necesidad de tener una herramienta completa de creación de escenarios (que estaba fuera del alcance del proyecto). Además, modelos tan detallados como este del coliseo tienen un valor educativo muy alto para enseñar a los estudiantes.

Otro atributo que podemos ver en la definición de la DTD de una escena es la **posición inicial por defecto**. Es una referencia a la posición del jugador en la escena cuando esta se cargue por primera vez. Configurando la posición inicial podemos asegurarnos de que el jugador no se colocará sobre una pared, o en el interior de un objeto cuando haya un cambio a esa escena. Es una posición en el espacio, así que viene definida por tres coordenadas.

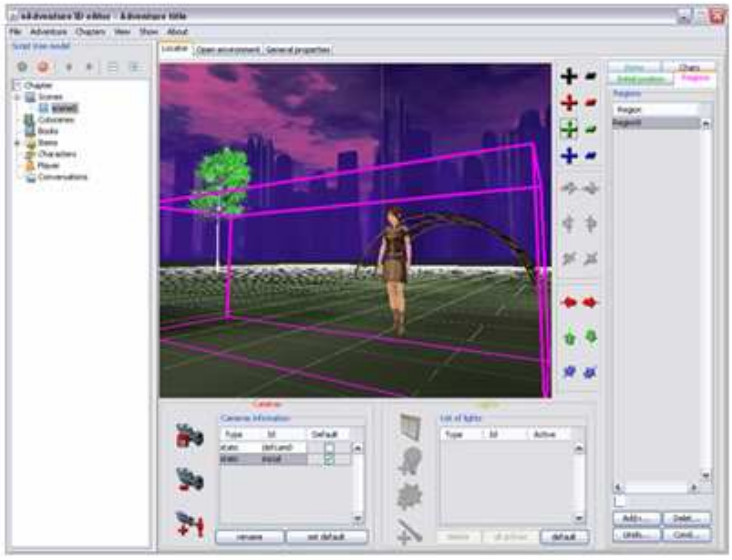
Hay otros dos atributos de la escena: **las referencias a objetos y personajes** que contienen respectivamente dos listas con los objetos y los personajes que aparecen en esa escena. Para cada objeto y personaje de la lista, también se fija su posición inicial en la escena. Por otra parte, si es necesario hacer transformaciones se pueden aplicar a estas referencias. Esto es muy útil porque puede haber más de un elemento repetido en la escena con diversos tamaños o rotaciones. Es también útil transformar las referencias en vez de hacer estos cambios al elemento original porque es más fácil de transformar mientras que estamos previsualizando la escena entera usando la herramienta de edición de juegos. Hablamos sobre transformaciones de una manera más detallada en la sección 2.3.4 de este capítulo.

Otra parte de la definición de las escenas son las **regiones**, que son totalmente opcionales. Cuando utilizamos los cuartos cerrados se le impide al personaje principal caminar a través de las paredes porque el sistema de control de las colisiones detecta el movimiento dentro de objetos físicos e impide al personaje

principal entrar en ellos. Sin embargo, en los ambientes abiertos donde no hay paredes el personaje principal puede caminar a dondequiera que el usuario desee (aunque tampoco puede atravesar objetos). En el primer caso esta es la razón por la cual decidíamos crear el concepto de región, pero son útiles para muchos otros propósitos. Al principio definimos las regiones como prismas rectangulares transparentes en la escena que el personaje principal no puede atravesar. Después ampliamos el concepto, permitiendo que también haya regiones que el jugador puede atravesar. Si una región puede ser atravesada por el jugador nos es útil porque podemos saber donde está y, por ejemplo, cambiar la cámara o encender la luz (es decir, lanzar cualquier efecto). A continuación pondremos la definición de una región junto con un ejemplo para un espacio abierto que ya conocemos:

```
<!--REGIONS-->
<!ELEMENT regions (region+)>
<!ELEMENT region (documentation?, condition?, effect?, post-effect?, transformations?)>
<!ATTLIST region
   id CDATA #REQUIRED
   %positionR;
   crossable (yes | no) "yes"
>
```

```
<regions>
  <region crossable="no" id="Region0" x="90" y="0" z="228">
    <transformations>
      <objectScale>
        <scale-axis-x scale="1090.7407">
        <scale-axis-y scale="394.46466">
        <scale-axis-z scale="340.75345">
      </objectScale>
    </transformations>
  </region>
</regions>
```



Hemos añadido una región a la escena que no se puede atravesar. Su función es impedir que el jugador vaya extremadamente cerca de la cámara o incluso se salga de la vista abarcada por la cámara. Las regiones se dibujan en rosa en el editor para que los usuarios puedan modificarlas, pero son transparentes en el motor.

La sección de la **vista** de la escena en la DTD hace referencia a la definición de cámaras y luces. Esta sección es la parte más complicada del lenguaje.

Al principio de esta sección escribimos sobre la dificultad de guardar la simplicidad del lenguaje <e-Adventure2D> por culpa de extensiones como las cámaras. En <e-Adventure2D> las escenas se visualizaban siempre desde una

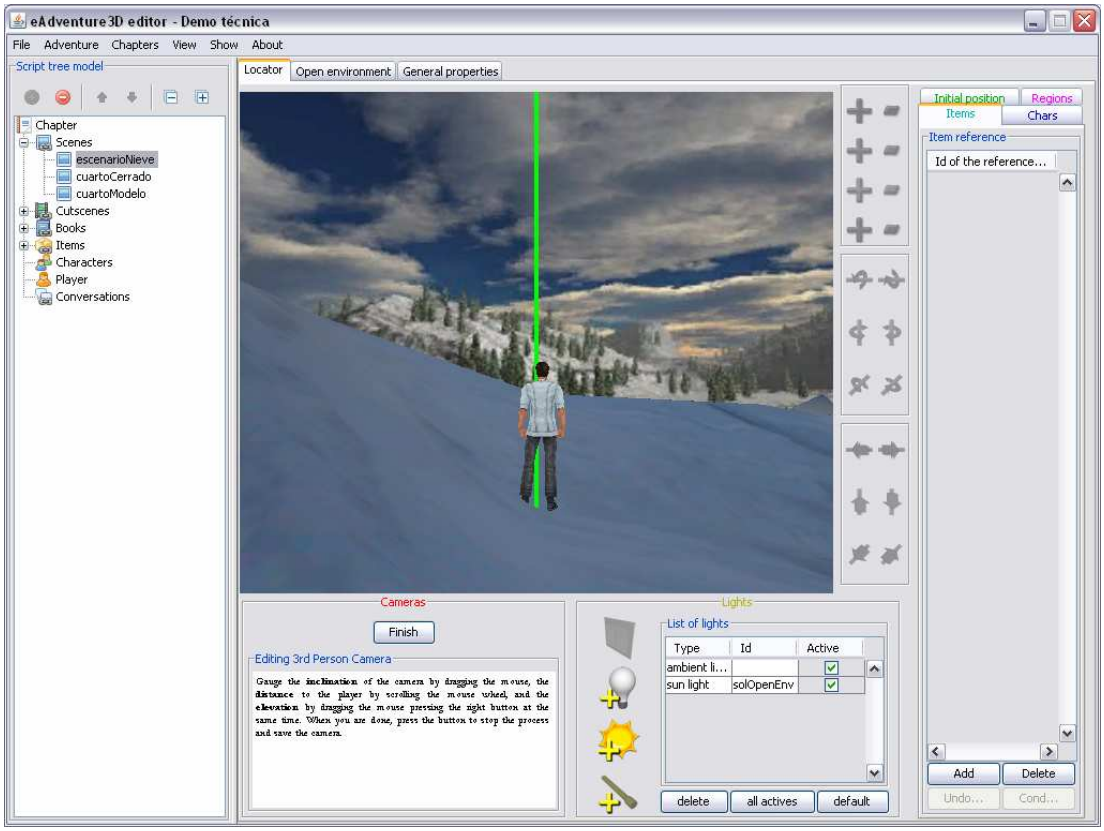
'cámara estática' situadas siempre en el mismo lugar (no es necesario el uso de cámaras en juegos en dos dimensiones). En un espacio tridimensional este planteamiento sería muy pobre, así que permitimos infinitos puntos de vista de una misma escena.

Permitimos dos clases de cámaras: cámaras estáticas y cámaras en tercera persona. A pesar de que hemos simplificado tanto como es posible los parámetros necesarios para definir una cámara, los fabricantes de juegos que no desean utilizar el editor deben conocer algunos conceptos de geometría básica. Ésta es la definición del lenguaje <e-Adventure3d> para las cámaras:

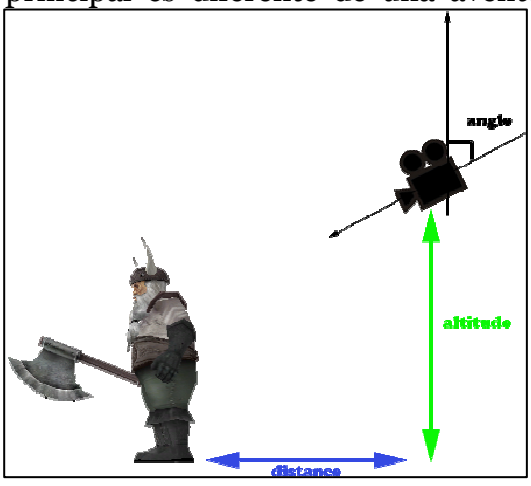
```
<!--CAMERAS-->
<!ELEMENT cameras (staticCamera|thirdPersonCamera)+>

<!--static camera-->
<!ELEMENT staticCamera (direction?, position?, documentation?)>
<!ATTLIST staticCamera
   id ID #REQUIRED
   default-camera (yes|no)  #IMPLIED
>
<!--third person camera-->
<!ELEMENT thirdPersonCamera (cameraData?, documentation?)>
<!ATTLIST thirdPersonCamera
   id ID #REQUIRED
   default-camera (yes|no) #IMPLIED
>
<!ELEMENT cameraData EMPTY>
<!ATTLIST cameraData
   angle NMTOKEN #REQUIRED
   distance NMTOKEN #REQUIRED
   altitude NMTOKEN #REQUIRED
>

<!ELEMENT direction EMPTY>

<!ATTLIST direction
   %positionR;
>
<!ELEMENT position EMPTY>
<!ATTLIST position
   %positionR;
>
```

Por una parte, tenemos las **cámaras en tercera persona** las cuales siguen al jugador haya donde se mueve a lo largo de una escena. La mejor manera de entender el funcionamiento es probar jugando pero podemos ver la manera de crear este tipo de cámaras para entender el concepto:

```
<thirdPersonCamera default-camera="no" id="chaseCamera">
   <cameraData altitude="14.101095" angle="1.4707963" distance="63.69816"/>
</thirdPersonCamera>
```



En el extracto de XML se muestran algunos de los parámetros que podemos calibrar en las cámaras en tercera persona por una sencilla razón: el personaje principal es diferente de una aventura a otra. Los usuarios tendrán que poder definir cuánto de cerca estará la cámara del jugador (atributo distancia). Por la misma razón, la altura de unos personajes será distinta a la de otros así que habrá que indicar la altura de la cámara, medida desde el suelo (o el terreno si estamos en un espacio abierto). Finalmente, los usuarios pueden ajustar el ángulo de la inclinación en radianes (respectivos al plano de XZ en el cual se ponen todos los elementos) de esta forma podemos variar el punto de vista desde el que observamos al jugador. La figura de la izquierda representa esta idea.



Por otra parte, existe la posibilidad de establecer tantas **cámaras estáticas** como queramos en una escena. Están definidas por la dirección a donde la cámara apunta y su posición en el espacio. Se ha simplificado la manera de definir una cámara estática de tal forma que se sitúan paralelas al plano XZ (girarlas no está permitido). Tal simplificación reduce el número de vectores requeridos para definir una cámara fotográfica a dos en lugar de tres.

Vamos a ver un ejemplo de cuatro cámaras estáticas distintas situadas en varios lugares de una misma escena:

```
<staticCamera default-camera="no" id="camaraTerraza3rd">
   <direction x="0.021410223" y="-0.3292196" z="0.9440106"/>
   <position x="-23.635534" y="117.033875" z="-347.555"/>
</staticCamera>
<staticCamera default-camera="no" id="entradaCamara">
   <direction x="-0.3638072" y="-0.50810176" z="-0.78069013"/>
   <position x="-37.459175" y="179.76706" z="25.949821"/>
</staticCamera>
<staticCamera default-camera="no" id="generalCam">
   <direction x="0.89513266" y="-0.31501454" z="0.3154418"/>
   <position x="-470.13004" y="226.8129" z="-245.78062"/>
</staticCamera>
<staticCamera default-camera="yes" id="inicialCam">
   <direction x="-0.27861503" y="-0.43314517" z="-0.8571808"/>
   <position x="0.45132333" y="78.20823" z="-244.35916"/>
</staticCamera>
```

La definición de cámaras estáticas en XML tiene un poco de miga. Como ya hemos indicado, los usuarios no sólo necesitan indicar la posición de la cámara, sino también el vector que defina a donde apunta. Intentar conjeturar la posición correcta de dichos vectores en el XML para que la cámara aparezca donde queremos al ejecutar la aventura en el motor puede llevar mucho tiempo, pero el editor ha simplificado este proceso más de lo que podíamos incluso imaginar en un

principio. Como ocurre en algunas herramientas de edición en tres dimensiones, los usuarios pueden mover el punto de vista de la escena con tan sólo arrastrar y soltar el ratón y la cámara en cada momento puede ser guardada automáticamente con tan solo pulsar un botón. Durante todo este proceso los usuarios no necesitan saber nada sobre los coordenadas de la cámara (es decir, los vectores) de hecho los usuarios del editor ni siquiera las ven.

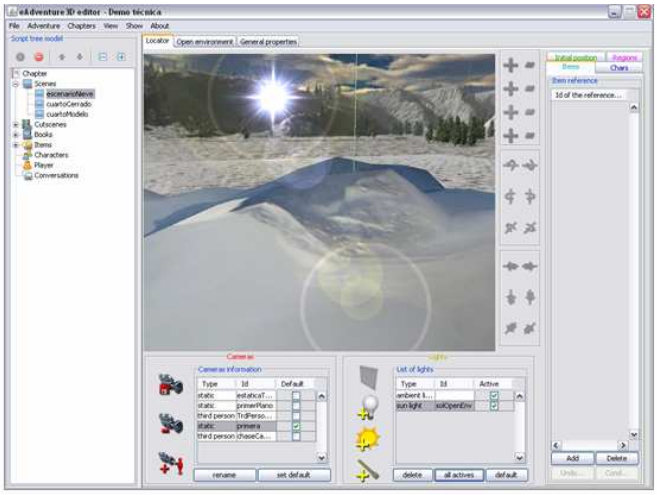
Agregar **luces** a las escenas también requiere un cierto conocimiento de geometría básica. Sin embargo, es más fácil definir las luces por defecto para cada tipo de escena que las cámaras, así que las luces son una parte opcional en el lenguaje <e-Adventure3D>. Contemplamos 4 tipos posibles de luces: una luz *ambiente*, una luz de *bombilla*, una luz de *linterna* y una luz similar a la luz del *sol*:

```
<!-- LIGHT-->
<!ELEMENT lights (ambient-light?, (bulb-light | lantern-light | sun-light)*)>
<!ELEMENT bulb-light (position,light-color)>
<!ATTLIST bulb-light
    id ID #REQUIRED
    active (yes|no) "yes"
>
<!ELEMENT lantern-light (position,direction,angle,light-color)>
<!ATTLIST lantern-light
    id ID #REQUIRED
    active (yes|no) "yes"
>
<!ELEMENT angle EMPTY>
<!ATTLIST angle
 degrees NMTOKEN #REQUIRED
>
<!ELEMENT sun-light (direction,light-color)>
<!ATTLIST sun-light
    id ID #REQUIRED
    active (yes|no) "yes"
>
<!ELEMENT ambient-light (light-color)>
<!ELEMENT light-color EMPTY>
<!ATTLIST light-color
   red NMTOKEN #REQUIRED
   green NMTOKEN #REQUIRED
   blue NMTOKEN #REQUIRED
   alpha NMTOKEN #IMPLIED
>
```

La **luz ambiente** se debe combinar con otras luces, así que veremos algunos ejemplos de las demás. En todos los tipos de luz tenemos que definir el color que tendrá.

El primer ejemplo es un ejemplo de **luz solar** en un espacio abierto. Para definir este tipo de luz se requiere definir la dirección de los rayos:

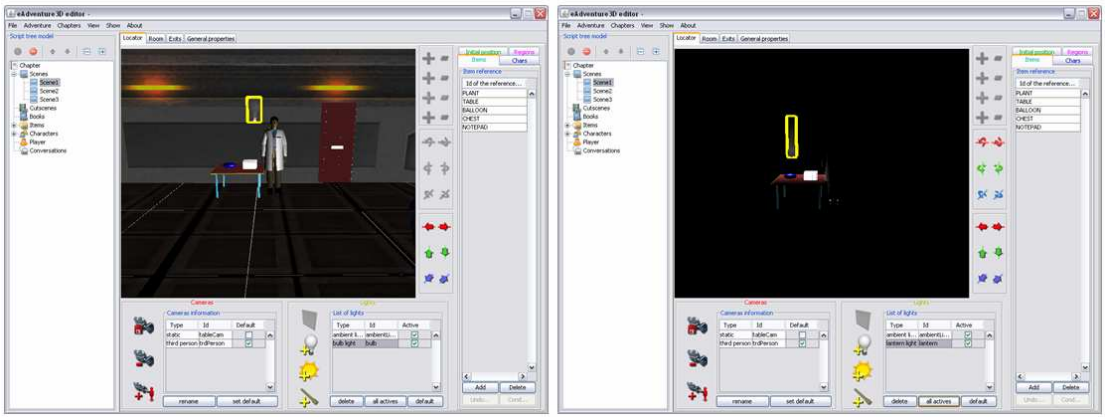
```
<sun-light active="yes" id="solOpenEnv">
   <direction x="-50.0" y="100.0" z="50.0"/>
   <light-color alpha="1.0" blue="1.0" green="1.0" red="1.0"/>
</sun-light>
```



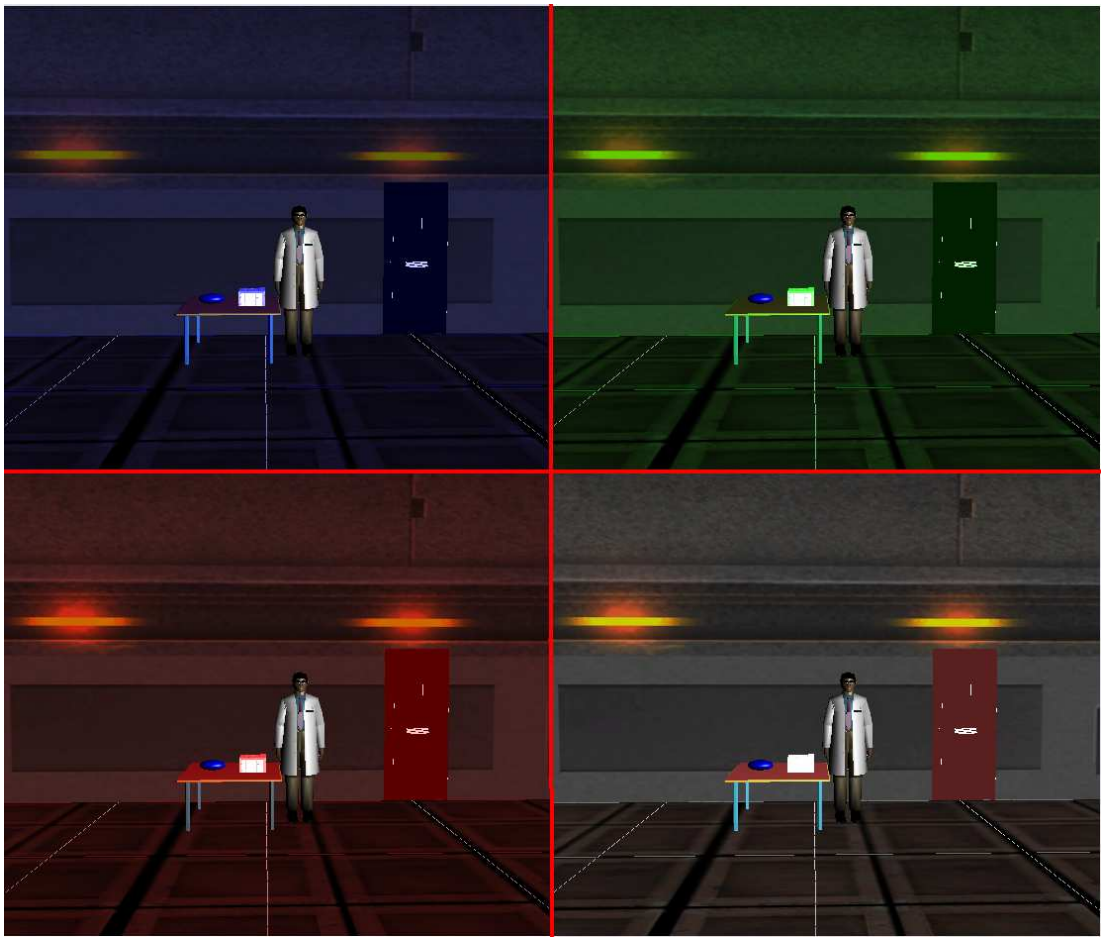
A continuación vamos a ver un mismo cuarto cerrado iluminado con dos tipos de luces diferentes. En la imagen izquierda hay una bombilla y en la derecha podemos ver una linterna. Como vemos, una **luz de bombilla** se define por su posición y una **luz de linterna** requiere además de una posición, la dirección a donde la luz apunta y el ángulo que cubre dicha luz.

```
<bulb-light active="yes" id="bulb">
   <position x="-25.0" y="24.0" z="-28.0"/>
   <light-color alpha="1.0" blue="1.0" green="1.0" red="1.0"/>
</bulb-light>
```

```
<lantern-light active="yes" id="lantern">
   <position x="-33.0" y="25.0" z="-45.0"/>
   <direction x="0.0" y="-1.0" z="8.940697E-8"/>
   <angle degrees="35.0"/>
   <light-color alpha="1.0" blue="1.0" green="1.0" red="1.0"/>
</lantern-light>
```



Para apreciar los efectos del color en una luz vamos a mostrar la misma escena anterior iluminada por la misma bombilla pero esta vez con color azul, verde, rojo y, por último, transparente:



Como observación final, discutiremos las ventajas que aportan a la plataforma los elementos de la vista (cámaras y luces). Por una parte, la combinación de cámaras y luces realza el poder expresivo de lo que se puede hacer con <e-Adventure3D>. Una buena combinación de luces y de cámaras hace al juego más atractivo desde un punto de vista visual, mejorando los rasgos de motivación del juego (que es una de las primeras razones por las que utilizar video juegos en la educación). Sin embargo, la ventaja principal no es la mejora de la apariencia en los juegos, la mayor ventaja recae en los rasgos educativos de estos elementos. Los usuarios pueden cambiar de cámara y cambiar luces encendiéndolas o apagándolas a lo largo del juego. Se pueden utilizar estas características con el fin de dirigir al alumno hacia el verdadero dominio del juego, es decir, sirven para dirigir su atención hacia elementos determinado. Se podría decir, por tanto, que son herramientas de gran alcance para mejorar el valor educativo de los juegos.

2.3.2 Escenas de corte

En segundo lugar vamos a explicar las escenas de corte (en inglés conocidas como 'cut-scenes'), el otro tipo de escenas permitidas en <e-Adventure3D>. Se

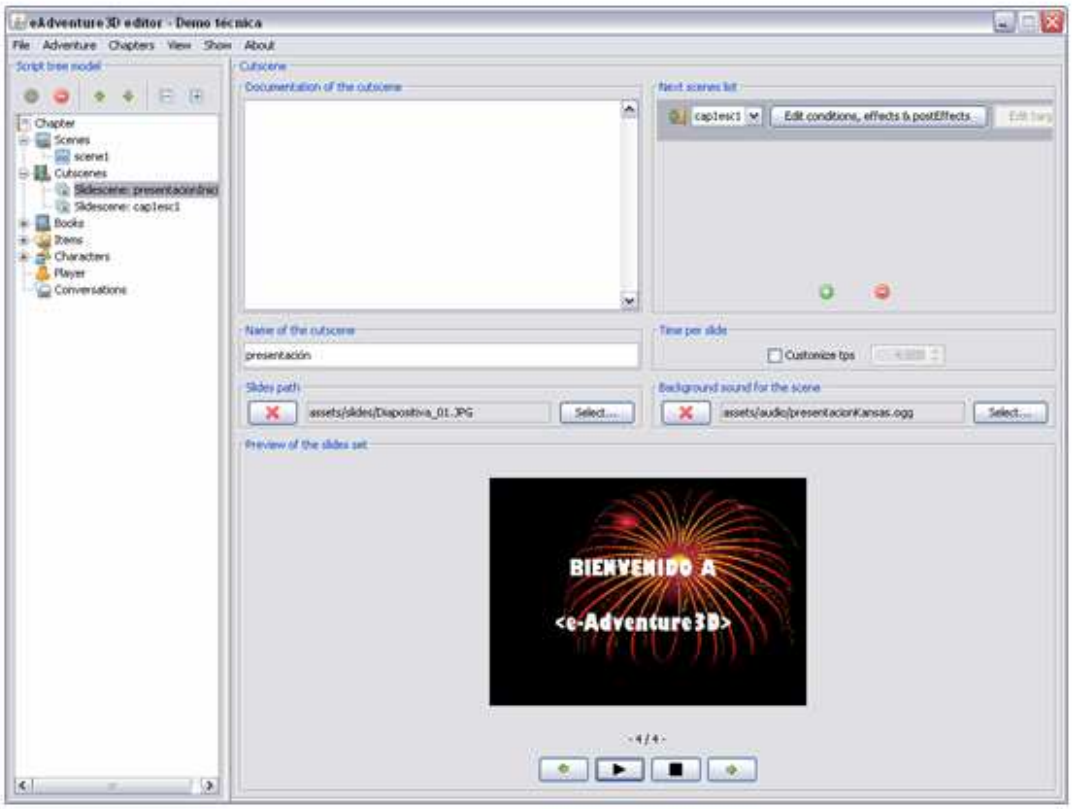
diferencian de las escenas normales en que los usuarios no pueden interactuar con ellas (por esa razón también son denominadas escenas no interactivas). Son muy útiles para mostrar información en juegos educativos. Hay dos tipos: *escenas de transparencias* y las *escenas de vídeo*.

Las **escenas de transparencias** (o diapositivas) son una sucesión de imágenes (diapositivas) que se muestran a pantalla completa. Para definirlas los usuarios tienen que enumerar las imágenes que aparecerán en la escena. También pueden fijar el tiempo entre imágenes y un sonido del fondo durante todo el tiempo. Este sonido de fondo se agrega como recurso (al igual que en las escenas interactivas) y por eso no aparece directamente en la definición de la DTD:

```
<!--SLIDESCENE-->
<!ELEMENT slidescene (documentation?, resources+, name, end-chapter? ,next-scene*)>
<!ATTLIST slidescene
   id ID #REQUIRED
   start (yes | no) "no"
   time-per-slide NMTOKEN #IMPLIED
>
```

Vamos a ver un ejemplo de escena de transparencias. Como siempre mostraremos el código XML del ejemplo y, en este caso, su resultado en el editor que será lo más ilustrativo:

```
<slidescene id="presentacionInicial" start="yes">
  <resources>
    <asset type="bgsound" uri="assets/audio/presentacionKansas.ogg"/>
    <asset type="slides" uri="assets/slides/Diapositiva_01.JPG"/>
  </resources>
  <name>presentación</name>
  <next-scene idTarget="cap1esc1"/>
</slidescene>
```



Todas las diapositivas de la carpeta ‘slides’ guardadas en el archivo EA3D con el nombre ‘Diapositiva_’ + número serán mostradas en la escena en el orden que indique el número.

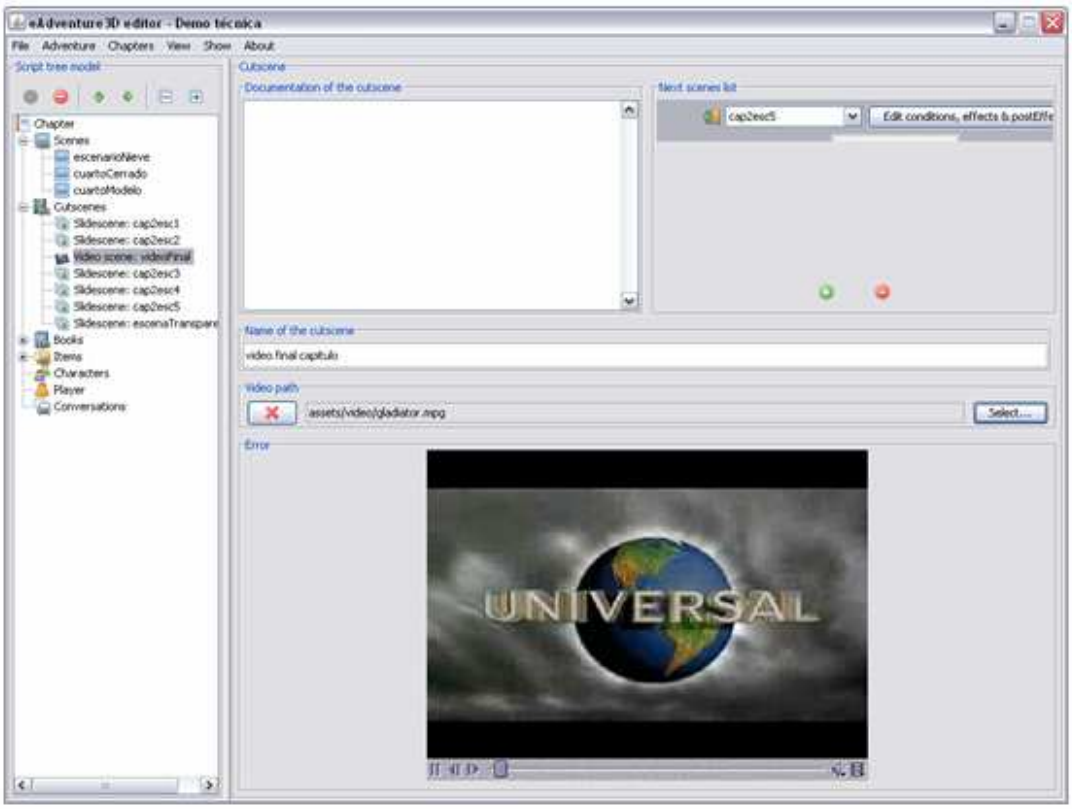
Las **escenas video** vienen definidas únicamente por un vídeo. En el motor, el vídeo será mostrado de principio a fin, así que una escena video durará lo que dure el video. En estas escenas no se permite añadir sonidos puesto que los videos pueden llevar sonido por si mismos:

```
<!--VIDEOSCENE-->
<!ELEMENT videoscene (documentation?, resources+, name, end-chapter?, next-scene*)>
<!ATTLIST videoscene
   id ID #REQUIRED
   start (yes | no) "no"
>
```

Como hemos hecho para las escenas de transparencias, vamos a poner un ejemplo de XML para los videos y su representación en el editor. Aprovechamos para explicar con este ejemplo el uso de la etiqueta ‘siguiente escena’ (en la DTD ‘next-scene’) a través de la cual se indica la siguiente escena que vendrá después

del video y la posición en la que se situará inicialmente el personaje principal en esa escena:

```
<videoscene id="videoFinal" start="no">
   <resources>
      <asset type="video" uri="assets/video/gladiator.mpg"/>
   </resources>
   <name>video final capitulo</name>
   <next-scene idTarget="cap2esc5" x="-1" y="-1" z="-1"/>
</videoscene>
```



Nuestra plataforma soporta los archivos de video de extensiones AVI, MOV y MPG. Sin embargo, hay que tener en consideración que hay muchas codificaciones distintas que se pueden aplicar a estos tipos de videos (sobretodo de extensión AVI). Por esa razón la visualización correcta de cierto vídeo depende de algunos factores como el sistema operativo, los codecs instalados, etc., incluso aunque el vídeo tenga una extensión de archivo reconocida por el motor. Hay que decir que el motor de java JME no permite videos, por tanto hemos usado librerías externas para conseguir que funcionen en nuestras aventuras.

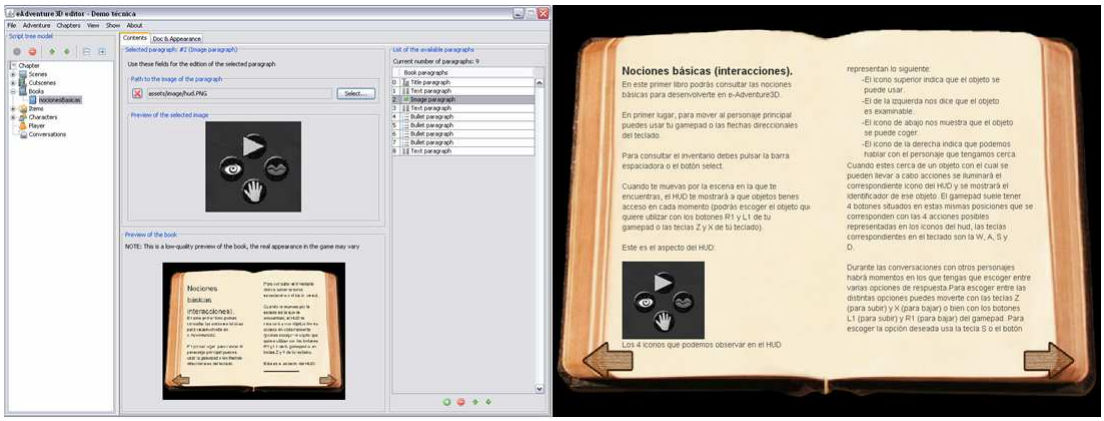
2.3.3 Libros

```
<!ELEMENT book (documentation?, resources+, text)>
<!ATTLIST book
   id ID #REQUIRED
>
<!ELEMENT text (#PCDATA | title | bullet | img)*>
<!ELEMENT bullet (#PCDATA)>
<!ELEMENT img EMPTY>
<!ELEMENT title (#PCDATA)>
<!ATTLIST img
   src CDATA #REQUIRED
>
```

Los **libros** son elementos que por lo general no suelen ser parte de los videojuegos. En nuestro caso su función es mejorar las características educativas de nuestras aventuras gráficas. Pueden ser proporcionados a los estudiantes cuando hay gran cantidad de información para ser transmitida o , simplemente, para que tengan una guía de referencia del tema a estudiar.

Como podemos ver en la DTD, los libros se editan a base de repetir cuatro tipos diferentes de párrafos (en el orden que se quiera y las veces que se quiera): título, texto, párrafos de viñeta y párrafos de imagen. Veamos un ejemplo para entender la diferencia entre ellos:

```
<book id="nocionesBasicas">
   <resources>
      <asset type="image" uri="assets/image/defaultbook.jpg"/>
   </resources>
   <text>
      <title>Nociones básicas (interacciones).</title>En este primer libro podrás consultar las nociones básicas para desenvolverte en e-Adventure3D.

En primer lugar, para mover al personaje principal puedes usar tu gamepad o las flechas direccionales del teclado.

Para consultar el inventario debes pulsar la barra espaciadora o el botón select.

Cuando te muevas por la escena en la que te encuentras, el HUD te mostrará a que objetos tienes acceso en cada momento (podrás escoger el objeto que quiere utilizar con los botones R1 y L1 de tu gamepad o las teclas Z y X de tú teclado).

Este es el aspecto del HUD:
      <img src="assets/image/hud.PNG"/>Los 4 iconos que podemos observar en el HUD representan lo siguiente:
      <bullet>El icono superior indica que el objeto se puede usar.</bullet>
      <bullet>El de la izquierda nos dice que el objeto es examinable.</bullet>
      <bullet>El icono de abajo nos muestra que el objeto se puede coger.</bullet>
      <bullet>El icono de la derecha indica que podemos hablar con el personaje que tengamos cerca.</bullet>Cuando estes cerca de un objeto con el cual se pueden llevar a cabo acciones se iluminará el correspondiente icono del HUD y se mostrará el identificador de ese objeto. El gamepad suele tener 4 botones situados en estas mismas posiciones que se corresponden con las 4 acciones posibles representadas en los iconos del hud, las teclas correspondientes en el teclado son la W, A, S y D.

Durante las conversaciones con otros personajes habrá momentos en los que tengas que escoger entre varias opciones de respuesta.Para escoger entre las distintas opciones puedes moverte con las teclas Z (para subir) y X (para bajar) o bien con los botones L1 (para subir) y R1 (para bajar) del gamepad. Para escoger la opción deseada usa la tecla S o el botón inferior de los 4 encionados en el gamepad.
   </text>
```



Estas dos capturas de pantalla muestran los resultados de cargar el anterior documento XML en el editor (a la izquierda) y su vista en el motor (en la imagen derecha)

2.3.4 Definiciones de elementos: el jugador, los personajes y los objetos

Algunas pequeñas diferencias se presentan a la hora de definir los personajes, el personaje principal y los objetos como podemos ver en la definición de la DTD de cada uno de ellos:

```
<!--OBJECT (ITEM)-->
<!ELEMENT object (documentation?, instance*, resources+, description, actions?, transformations?)>
<!ATTLIST object
   id ID #IMPLIED
>
<!--PLAYER-->
<!ELEMENT player (documentation?, resources+, textcolor?, description, characterDirection, characterUpVector, characterAnimations?, transformations?)>

<!--CHARACTER-->
<!ELEMENT character (documentation?, resources+, textcolor?, description, conversations?, characterDirection?,characterUpVector?,characterAnimations?,
transformations?)>
```

Vamos a explicar las características que comparten:

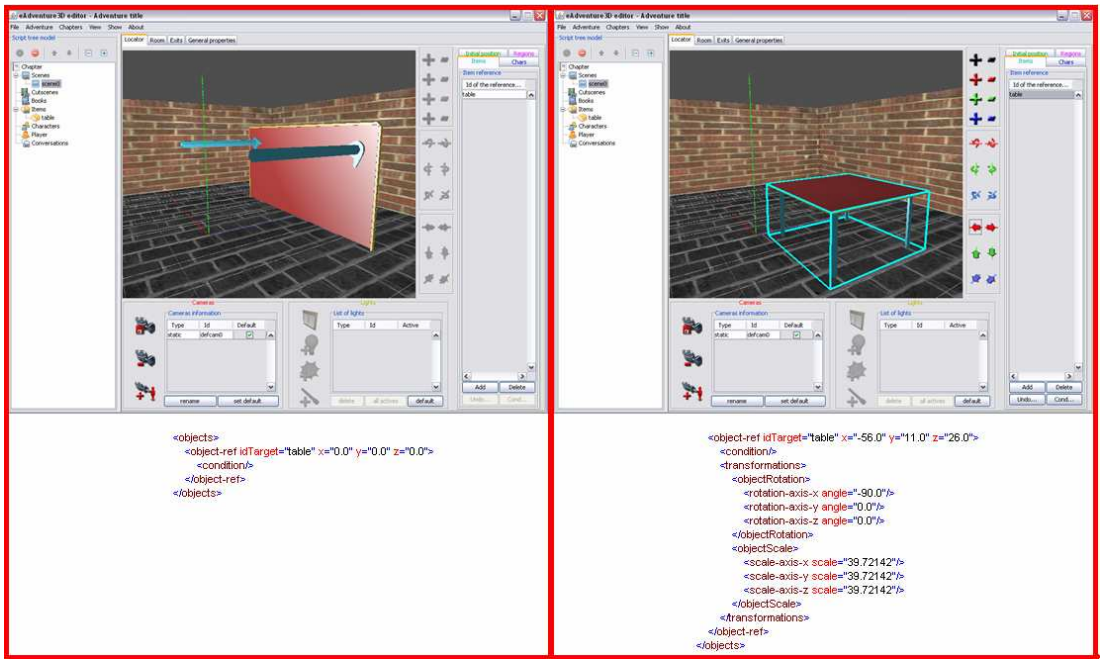
El bloque de los **recursos** contiene las referencias a los recursos artísticos usados en el 'renderizado' del elemento. Pueden incluir el lugar donde guardamos en el disco los siguientes archivos: el modelo 3D que representa el elemento, la textura que deseamos aplicar a dicho modelo y, en el caso de los objetos, el icono que lo representa (sólo es necesario en caso de que vaya a ser almacenado en el inventario del personaje principal). JME permite el uso de modelos 3d diseñados con diversos programas, las extensiones admitidas son: 3DS, ASE, DAE, JME, MD2, MD3, MS3D y OBJ.

Los modelos 3D que se cargarán en los juegos deberían ser diseñados acorde a la escena en la que se van a usar (con un tamaño y una orientación adecuada). Sin embargo, en caso de que no sea así, no hay ningún problema ya que permitimos

ciertas transformaciones que se pueden aplicar a los modelos tridimensionales: las transformaciones de escala cambian el tamaño del modelo y las de rotación se encargan de girar los modelos sobre cada uno de los 3 ejes de coordenadas. Así es como se definen las transformaciones en el lenguaje <e-Adventure3D>:

```
<!--TRANSFORMATIONS-->
<!ELEMENT transformations ((objectRotation | objectScale)+)>
<!ELEMENT objectScale (scale-axis-x?,scale-axis-y?,scale-axis-z?)>
<!ELEMENT objectRotation (rotation-axis-x?, rotation-axis-y?, rotation-axis-z?)>
<!ELEMENT rotation-axis-x EMPTY>
<!ELEMENT rotation-axis-y EMPTY>
<!ELEMENT rotation-axis-z EMPTY>
<!ATTLIST rotation-axis-x
   %angle;
>
<!ATTLIST rotation-axis-y
   %angle;
>
<!ATTLIST rotation-axis-z
   %angle;
>

<!ATTLIST scale-axis-x
   scale NMTOKEN #REQUIRED
>
<!ATTLIST scale-axis-y
   scale NMTOKEN #REQUIRED
>
<!ATTLIST scale-axis-z
   scale NMTOKEN #REQUIRED
>
<!ELEMENT scale-axis-x EMPTY>
<!ELEMENT scale-axis-y EMPTY>
<!ELEMENT scale-axis-z EMPTY>
```

Cuando es necesario las transformaciones se pueden aplicar a los modelos 3D que queremos usar en nuestras aventuras. Como hemos dicho, las transformaciones son muy útiles cuando alguien utiliza modelos que no habían sido diseñados para su propio proyecto o simplemente para reutilizar los mismos modelos en diversas situaciones. En la aventura de ejemplo utilizamos modelos gratis descargados de Internet que, como era de esperar, en principio no encajaban en nuestras escenas, pero aplicamos transformaciones a los modelos. Y tras estas transformaciones da la sensación de que nosotros mismos hubiésemos diseñado los modelos utilizados. Veamos un sencillo ejemplo:



En la imagen izquierda acabamos de cargar un modelo de una mesa. Lo hemos descargado de Internet así que no se ha diseñado para este juego y, como era de esperar, no esta acorde con el resto de la escena (sale girada y demasiado desproporcionada de tamaño). La imagen derecha es el resultado después de aplicar algunas transformaciones a la referencia del artículo en la escena: se ha escalado el modelo sobre los tres ejes de modo que ha quedado reducido al 39%; también ha sido rotado noventa grados sobre el x y, por último, hemos cambiado su posición en la escena (ya que por defecto aparece en la posición (0, 0, 0)). Este proceso es bastante arduo si se hace directamente sobre el XML ya que hay que llevar a cabo un proceso cíclico de probar en el motor y modificar sobre el lenguaje hasta alcanzar el resultado buscado; sin embargo, gracias al editor, los usuarios no deben preocuparse ya que todo se hace gráficamente gracias a la barra de transformaciones (resaltada e la imagen derecha) y a la pantalla de previsualización.

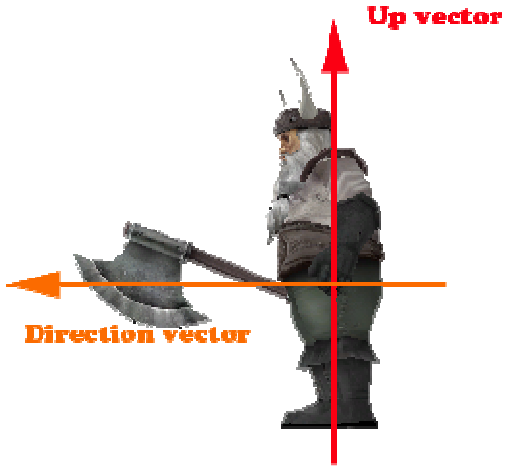
Los personajes y el personaje principal tienen cuatro atributos adicionales en su parte de definición del lenguaje:

- **Color del texto ('text colour')**: esta cualidad debe ser especificada si vamos a utilizar conversaciones con subtítulos en las que participe el personaje que estamos definiendo; ya que este atributo es el color de los subtítulos para este personaje. Es útil para distinguir que personaje esta hablando en conversaciones en la que participen varios personajes.
- **Vector ascendente del personaje ('Character up vector'):** define el eje al cual es paralela la 'espina dorsal' del personaje en forma de

vector, el cual estará orientado hacia la 'cabeza' del personaje.

- **Dirección del personaje ('character direction')**: este vector indica al motor hacia donde mira el personaje en cuestión. Este vector debe ser normal a la 'cara' del personaje.

  Conjuntamente con el vector ascendente, ambos vectores dan al motor la información requerida para producir el movimiento del personaje y para la definición de las cámaras en tercera persona. Sin ellos podría suceder que el personaje se moviese "al revés" que no es, obviamente, el comportamiento esperado; aunque el personaje sea Michael Jackson.

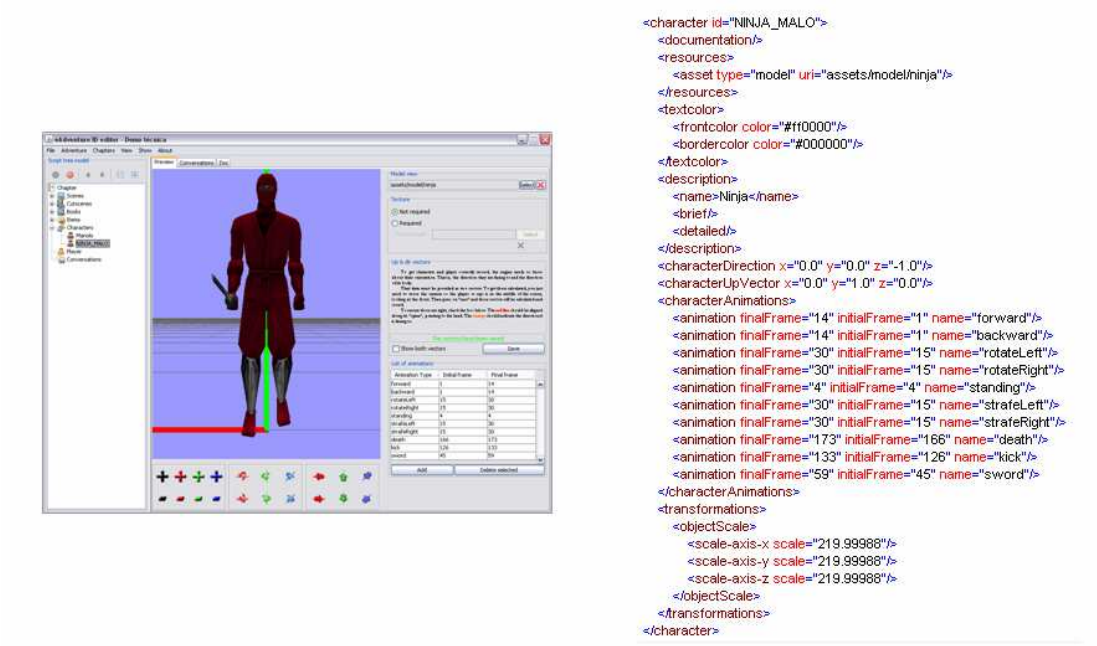


- Algunos modelos tridimensionales tienen asociadas todas sus **animaciones** seguidas como si de un video se tratase. Por lo tanto, los usuarios tienen que especificar cuál es el 'frame' inicial y el 'frame' final para cada animación, y dar un nombre para ella con el fin de que pueda ser referenciada fácilmente (no se puede sacar automáticamente cada animación ya que pueden estar grabadas entre cualesquiera 'frames' y además cada modelo puede tener unas distintas):

```
<!ELEMENT animation EMPTY>
<!ELEMENT characterAnimations (animation+)>
<!ATTLIST animation
   name CDATA #REQUIRED
   initialFrame NMTOKEN #REQUIRED
   finalFrame NMTOKEN #REQUIRED
>
```

Los personajes también tienen el atributo llamado **conversaciones** en el cual se guardan las referencias a las conversaciones entre dicho personaje y el personaje principal.

Vamos a ver un ejemplo de la definición completa de un personaje:



La parte izquierda de la imagen muestra el código XML (de la parte derecha) representado en el editor. A parte la edición de personajes en el editor consta de otras dos pestañas (como se puede apreciar en la imagen), una con la lista de conversaciones de ese personaje y otra con la documentación y el color del texto.

En referencia a los artículos hay un atributo extra llamado **acciones**. Este atributo guarda la lista de acciones que se pueden hacer con el objeto que estamos especificando. Hay cuatro acciones posibles representadas en el HUD:

- Coger (grab: en la parte sur del HUD): indica al usuario que el objeto se puede coger.
- Examinar (examine: en el oeste del HUD): indica que se puede examinar.
- Usar (use: en el norte del HUD): indica que el objeto se puede usar.
- Usar con (use with: en la parte norte del HUD): una vez que el objeto está en el inventario, esta acción del HUD indica que se puede usar con otros objetos (del propio inventario o de fuera) o personajes.

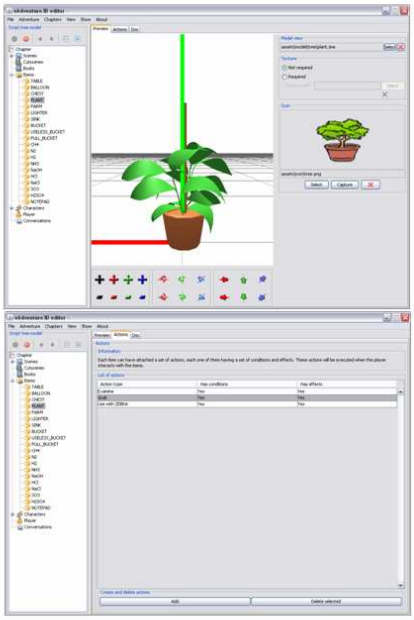
Estas acciones se pueden realizar con el objeto seleccionado en el momento. Mostramos dichas acciones remarcadas en el HUD:

El icono en la parte este del HUD se reserva a los personajes e indica si el

personaje accesible en ese momento por el jugador tiene conversaciones disponibles.

Las acciones proporcionan la interacción en <e-Adventure3D>. Las acciones disponibles para cierto objeto en cierto momento se sujetan a la satisfacción de ciertas condiciones y permiten la ejecución de un bloque de efectos (esto se detallará más adelante en el apartado de efectos).

Veamos un ejemplo de la edición de un objeto y el XML generado:




```
<object id="PLANT">
  <documentation/>
  <resources>
    <asset type="model" uri="assets\model\tree\plant.jme"/>
    <asset type="icon" uri="assets\icon\tree.png"/>
  </resources>
  <description>
    <name>The amazing tree</name>
    <brief>?</brief>
    <detailed>?</detailed>
  </description>
  <actions>
    <examine>
      <condition>
        <active flag="scene1Active"/>
      </condition>
      <effect>
        <trigger-conversation idTarget="examinePlant1"/>
      </effect>
    </examine>
    <grab>
      <condition>
        <active flag="scene1Active"/>
      </condition>
      <effect>
        <generate-object idTarget="PLANT"/>
        <activate flag="plantGrabbed"/>
      </effect>
    </grab>
    <use-with idTarget="ZEBRA">
      <condition>
        <active flag="usedBalloonWithZebra"/>
      </condition>
      <effect>
        <generate-object idTarget="PLANT"/>
        <activate flag="plantGrabbed"/>
      </effect>
    </grab>
    <use-with idTarget="ZEBRA">
      <condition>
        <active flag="usedBalloonWithZebra"/>
      </condition>
      <effect>
        <consume-object idTarget="PLANT"/>
        <activate flag="usedPlantWithZebra"/>
        <trigger-conversation idTarget="plantUsedWithZebra"/>
        <change-camera idTarget="zebraBottom"/>
        <trigger-conversation idTarget="balloonInflated"/>
        <change-camera idTarget="scopeCamera"/>
        <generate-object idTarget="CH4"/>
      </effect>
    </use-with>
  </actions>
  <transformations>
    <objectRotation>
      <rotation-axis-x angle="270.0"/>
      <rotation-axis-y angle="0.0"/>
      <rotation-axis-z angle="0.0"/>
    </objectRotation>
    <objectScale>
      <scale-axis-x scale="10.0"/>
      <scale-axis-y scale="10.0"/>
      <scale-axis-z scale="10.0"/>
    </objectScale>
  </transformations>
</object>
```

En la primera pestaña podemos ver una previsualización del modelo asociado al objeto tal y como se verá en el motor (en la imagen de arriba). También podemos ver el icono asociado generado automáticamente. En la segunda pestaña (imagen de abajo) vemos las acciones asociadas a este objeto. Hay una pestaña más en la edición de objetos destinada a la documentación.

2.3.5. Conversaciones

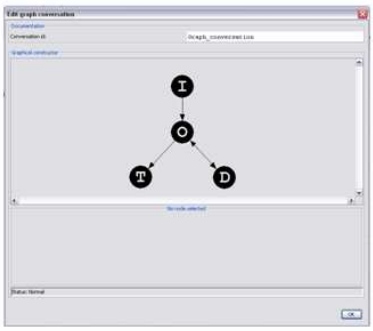
Las conversaciones son muy importantes en aventuras gráficas, pues proveen de

capacidad de interacción a los personajes y de una buena fuente de información para indicarle al jugador por donde seguir. Estos aspectos son muy interesantes para los juegos educativos, pues pueden ser utilizados para dirigir al estudiante durante la experiencia de aprendizaje y para proporcionarle información (son una manera entretenida de explicar cosas al estudiante).

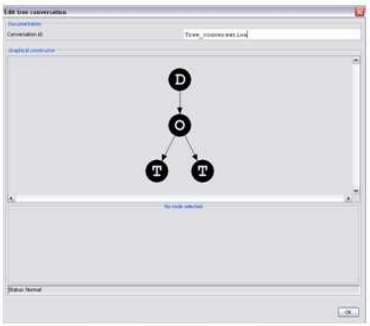
Hay dos tipos de conversaciones en <e-Adventure3D>: conversaciones de tipo grafo y conversaciones de tipo árbol. Ambos tipos son muy similares, sólo se diferencian en los posibles caminos que pueden tomar en cada caso. Primero veamos la definición de la DTD para cada uno de los tipos y, en segundo lugar, mostraremos un ejemplo donde podemos ver el fragmento de documento XML para los dos tipos de conversación junto con su representación en la herramienta de edición de juegos::

```
<!--CONVERSATION-->

<!ENTITY % conversation "tree-conversation | graph-conversation">
<!ENTITY % continuation "(response|end-conversation)">

<!--TREE CONVERSATION-->
<!ELEMENT tree-conversation (%speak;, %continuation;)>
<!ATTLIST tree-conversation
   id ID #REQUIRED
>
<!--GRAPH CONVERSATION-->
<!ELEMENT graph-conversation (dialogue-node | option-node)+>
<!ATTLIST graph-conversation
   id ID #REQUIRED
>
<!ELEMENT dialogue-node ((effect?,sound?,(speak-player | speak-npc))*, (child | end-conversation))>
<!ATTLIST dialogue-node
   nodeindex CDATA #REQUIRED
>
<!ELEMENT option-node (effect?,sound?,speak-player , child)+>
<!ATTLIST option-node
   nodeindex CDATA #REQUIRED
>
<!ELEMENT child EMPTY>
<!ATTLIST child
   nodeindex CDATA #REQUIRED
>
<!ELEMENT sound EMPTY>
<!ATTLIST sound
   url CDATA #REQUIRED
>
<!ELEMENT response (option)+>
<!ELEMENT option (effect?, sound?,speak-player, %speak;, (%continuation; | go-back))>
<!ELEMENT go-back EMPTY>
<!ELEMENT end-conversation (effect?)>
```

```
<graph-conversation id="Graph_conversation">
  <dialogue-node nodeindex="0">
    <speak-player>Hi, how are you?</speak-player>
    <child nodeindex="1"/>
  </dialogue-node>
  <option-node nodeindex="1">
    <speak-player>I'm OK</speak-player>
    <child nodeindex="2"/>
    <speak-player>I'm not OK</speak-player>
    <child nodeindex="3"/>
  </option-node>
  <dialogue-node nodeindex="2">
    <speak-npc idTarget="TEACHER">I'm glad to hear that</speak-npc>
    <end-conversation/>
  </dialogue-node>
  <dialogue-node nodeindex="3">
    <speak-player>But I have good news for you, you have passed all your exams. How are you now?</speak-player>
    <child nodeindex="1"/>
  </dialogue-node>
</graph-conversation>
```




```
<tree-conversation id="Tree_conversation">
  <speak-player>Hi!</speak-player>
  <speak-npc idTarget="TEACHER">Hi!</speak-npc>
  <speak-npc idTarget="TEACHER">Do you want to start the class?</speak-npc>
  <response>
    <option>
      <speak-player>No.</speak-player>
      <speak-npc idTarget="TEACHER">Then go home</speak-npc>
      <end-conversation/>
    </option>
    <option>
      <speak-player>I'm not fine today.</speak-player>
      <speak-player>Then go home</speak-player>
      <end-conversation/>
    </option>
  </response>
</tree-conversation>
```



Las conversaciones se definen en términos de nodos. Aunque la idea puede ser difícil para los instructores (es decir, los fabricantes del juego) que no tienen conocimientos de programación. El concepto de nodo es necesario para que el motor soporte conversaciones en las que los usuarios participan activamente eligiendo entre varias líneas de diálogo para continuar la conversación (sólo en ciertos puntos). Esto es una característica común en juegos clásicos de aventuras gráficas, y sin esta característica las conversaciones serían monótonas y aburridas y, como consecuencia, inútiles para un propósito educativo. De este modo, hay dos tipos básicos de *nodos*: de *diálogo* y de *opción*. El primer tipo fue ideado para contener las líneas de diálogo de una conversación. Cuando se quiere que un usuario pueda elegir entre varias opciones, se debe utilizar un nodo de opción y ligarlo al nodo anterior del diálogo. Entonces, cada opción definida en el nodo de opción conducirá a un nodo de diálogo distinto, y de esta manera, la conversación tomará caminos muy distintos dependiendo de la opción escogida.

Principalmente, la diferencia entre los dos tipos de conversaciones es que las conversaciones de grafo permiten ciclos mientras que las conversaciones de tipo árbol no (como podemos observar en la imagen anterior). Las conversaciones de árbol tienen una sintaxis más fácil porque son siempre lineales. Sin embargo, todas las conversaciones de árbol se pueden escribir como conversaciones de grafo.

2.3.6 Indicadores, condiciones y efectos

Los **indicadores** (o 'flags') son booleanos que determinan el estado del juego en cada momento. Son como semáforos que pueden estar en verde o en rojo. De hecho, al igual que los semáforos controlan el tráfico permitiendo que los vehículos pasen cuando están en verde, los indicadores controlan el flujo narrativo de los juegos en <e-Adventure3D> ya que pueden estar en estado 'activo' o 'inactivo'.

Como hemos dicho, los indicadores pueden estar activados o desactivados y dependiendo de su estado el juego tomará una dirección u otra. De este modo, la función de los indicadores es que sean utilizados por los fabricantes del juego para establecer el curso del juego.

La pregunta es: ¿cómo se usan los indicadores? Se utilizan a través de las **condiciones**. Las condiciones se utilizan en distintos momentos del juego:

- Al cambiar de escena: antes de que el jugador pueda ir a otra escena, puede haber algunas condiciones que deban ser verificadas.
- En regiones: en regiones que el jugador puede atravesar puede haber condiciones que deban ser verificadas antes de que los efectos de la región puedan ser lanzados. (Los efectos se explican más adelante).
- En las acciones asociadas a los objetos y en las conversaciones: a veces es muy útil utilizar condiciones con las acciones asociadas a un objeto. Lo mismo sucede con las referencias de la conversación asociadas a un carácter. De este modo podemos definir varias acciones del mismo tipo o varias conversaciones con un mismo personaje y el motor escogerá aquella que satisfaga todas las condiciones. Así se soportan varios comportamientos conforme el juego va avanzando.
- En las referencias de una escena: las referencias a personajes y objetos de una escena pueden tener condiciones, de modo que estos elementos aparecerán en la escena sólo cuando se verifiquen sus condiciones.

Las condiciones se definen como vemos a continuación:

```
<!--CONDITION-->
<!ENTITY % basic-condition "(active|inactive)">
<!ELEMENT condition (%basic-condition; | either)*>
<!ELEMENT active EMPTY>
<!ATTLIST active
   flag NMTOKEN #REQUIRED
>
<!ELEMENT inactive EMPTY>
<!ATTLIST inactive
   flag NMTOKEN #REQUIRED
>
<!ELEMENT either (%basic-condition;)+>
```

Y ahora la pregunta es, ¿cómo se puede cambiar el estado de los indicadores a lo largo del juego? A través de los **efectos**.

Los efectos son acciones automáticas que pueden ser lanzados en varias partes de un juego:

- Al cambiar de escena: cuando la escena esta a punto de cambiar algunos efectos se pueden lanzar en la escena actual (efectos normales) y algunos otros efectos se pueden lanzar en la escena siguiente (conocidos como efectos posteriores).
- En regiones: cuando el jugador camina a través de una región algunos efectos se pueden lanzar a la entrada y otros pueden ser lanzados cuando el jugador abandona la región.
- En las acciones asociadas a los objetos: se pueden lanzar efectos cuando se lleva a cabo una acción asociada a un objeto.
- En las conversaciones: durante una conversación, los efectos se pueden lanzar después de cada línea (y cuando terminen de ejecutarse, la conversación continuará normalmente por donde iba) y también al finalizar la conversación.

Como se puede observar, los efectos se utilizan principalmente en los mismos lugares que las condiciones. Esto suele ser porque la mayoría de los efectos se lanzan después de verificar que ciertas condiciones se satisfacen (o sea comprobar que el estado actual del juego es el que corresponde).

Como podemos ver en el extracto siguiente de la DTD, los efectos no sólo se utilizan para cambiar los indicadores del juego sino que hay gran cantidad de efectos distintos:

```
<!--EFFECT-->
<!ENTITY % effects "(activate | deactivate | consume-object | generate-object | play-sound | cancel-action | speak-player | speak-npc | set-player-animation |
set-npc-animation | move-player | move-npc | change-light | change-camera | trigger-conversation | trigger-cutscene |  trigger-next-scene | trigger-book |
trigger-end-chapter)*">
<!ELEMENT effect (%effects;)>
<!ELEMENT post-effect (%effects;)>
<!ELEMENT activate EMPTY>
<!ATTLIST activate
   flag NMTOKEN #REQUIRED
>
<!ELEMENT deactivate EMPTY>
<!ATTLIST deactivate
   flag NMTOKEN #REQUIRED
>
<!ELEMENT consume-object EMPTY>
<!ATTLIST consume-object
   idTarget IDREF #REQUIRED
>
<!ELEMENT generate-object EMPTY>
<!ATTLIST generate-object
   idTarget IDREF #REQUIRED
>
<!ELEMENT play-sound EMPTY>
<!ATTLIST play-sound
   background (yes | no) "yes"
   uri CDATA #REQUIRED
>
<!ELEMENT cancel-action EMPTY>

<!ELEMENT speak-player (#PCDATA)>
<!ATTLIST speak-player
   uri CDATA #IMPLIED
>
<!ELEMENT speak-npc (#PCDATA)>
<!ATTLIST speak-npc
   idTarget IDREF #REQUIRED
   uri CDATA #IMPLIED
>
<!ELEMENT set-player-animation EMPTY>
<!ATTLIST set-player-animation
   animationId CDATA #REQUIRED
   repeat (yes | no) "no"
>
<!ELEMENT set-npc-animation EMPTY>
<!ATTLIST set-npc-animation
   characterId IDREF #REQUIRED
   animationId CDATA #REQUIRED
   repeat (yes | no) "no"
>

<!ELEMENT move-player (target-point+)>
<!ELEMENT move-npc (target-point+)>
<!ATTLIST move-npc
   idTarget IDREF #REQUIRED
>
<!ELEMENT target-point EMPTY>
<!ATTLIST target-point
   x CDATA #REQUIRED
   z CDATA #REQUIRED
>

<!ELEMENT change-light EMPTY>
<!ATTLIST change-light
   idTarget IDREF #REQUIRED
>
<!ELEMENT change-camera EMPTY>
<!ATTLIST change-camera
   idTarget IDREF #REQUIRED
>
<!ELEMENT trigger-conversation EMPTY>
<!ATTLIST trigger-conversation
   idTarget IDREF #REQUIRED
>

<!ELEMENT trigger-cutscene EMPTY>
<!ATTLIST trigger-cutscene
   idTarget IDREF #REQUIRED
>

<!ELEMENT trigger-next-scene EMPTY>
<!ATTLIST trigger-next-scene
   idTarget IDREF #REQUIRED
>

<!ELEMENT trigger-book EMPTY>
<!ATTLIST trigger-book
   idTarget IDREF #REQUIRED
>

<!ELEMENT trigger-end-chapter EMPTY>
```

La tabla siguiente resume los diferentes tipos de efectos. Para cada efecto indica

los parámetros necesarios para lanzarlo y un resumen de las funciones que lleva a cabo el efecto en cuestión:

Nombre del efecto	Parámetros	Función
Activar	El identificador del indicador	Activa un indicador
Desactivar	El identificador del indicador	Desactiva un indicador
Consumir objeto	El identificador de un objeto	Quita el objeto del inventario
Generar objeto	El identificador de un objeto	Genera el objeto en el inventario
Emitir sonido	El path donde tenemos guardado el sonido. También hay que indicar si el sonido se repite hasta cambiar de escena o no.	Hace sonar el sonido. Sonará más alto si no tiene que repetirse.
Habla el jugador	La frase que queremos que diga el jugador, escrita entre las marcas de XML y el path del sonido asociado a dicha frase (si se quiere).	Escribe la frase en la pantalla como subtítulo y ejecuta el sonido asociado al mismo tiempo.
Habla un personaje	Se necesita lo mismo que para el efecto anterior y además el identificador del personaje.	Hace lo mismo que el efecto anterior sólo que escribirá el subtítulo con el color asociado a este personaje.
Poner animación al jugador	El identificador de la animación y el atributo que indica si la animación debe repetirse hasta cambiarla por otra.	El jugador hará lo que tenga guardado en esa animación.
Poner animación a un personaje	Lo mismo que en el efecto anterior más el identificador del personaje.	El personaje hará lo que tenga guardado en esa animación.
Mover al jugador	La lista de puntos (sólo las	El jugador andará pasando por

	coordenadas X y Z) que queremos que recorra el jugador.	todos los puntos indicados, esquivando personajes y objetos.
Mover a un personaje	Lo mismo que en el anterior además del identificador del personaje.	Hace lo mismo que el anterior sólo que también esquivará al jugador si se cruza en su camino.
Cambiar luz	El identificador de la luz.	si la luz referenciada está encendida entonces se apagará y si está apagada se encenderá.
Cambiar cámara	El identificador de la cámara.	Cambia la cámara a la cámara referenciada, es decir, cambia el punto de vista de la escena.
Lanzar conversación	el identificador de la conversación	Lanza la conversación referenciada.
Cambiar escena	El identificador de la escena	Cambia la escena actual por la referenciada. También cambia la cámara por la que tenga por defecto la nueva escena.
Lanzar escena de corte	El identificador de una escena de video o de una escena de transparencias.	Lanza la escena de corte referenciada.
Mostrar libro	El identificador del libro.	Muestra el libro referenciado.
Cambiar de capítulo		Cambia al siguiente capítulo de acuerdo a su orden en el descriptor (cambia a la escena inicial del siguiente capítulo con la cámara que tenga esta por defecto). si no hubiese más capítulos, entonces el juego termina.

2.3.7 Evaluación automática y adaptación

Con todas las características explicadas hasta ahora los usuarios pueden producir aventuras gráficas completas de gran alcance. Sin embargo, su valor educativo sería limitado y es un aspecto que no se puede dejar a un lado. Los libros, las escenas de corte, las conversaciones, las acciones, las cámaras y las luces pueden mejorar el valor educativo de los juegos pero los instructores necesitan saber si la experiencia de aprendizaje es acertada (es decir, saber si el aprendiz alcanza las metas educativas propuestas por el instructor). En otro caso los instructores se verían obligados a hacer exámenes antes y después de que el alumno juegue al juego. Con este fin se utiliza la denominada *evaluación automática* ('assessment').

Por otra parte, no todos los estudiantes muestran las mismas capacidades ni tienen el mismo conocimiento del tema en estudio. Por estas razones es recomendable adaptar el comportamiento del juego dependiendo de quién está jugando; eso es lo que llamamos *adaptación* ('adaptation').

Evaluación automática

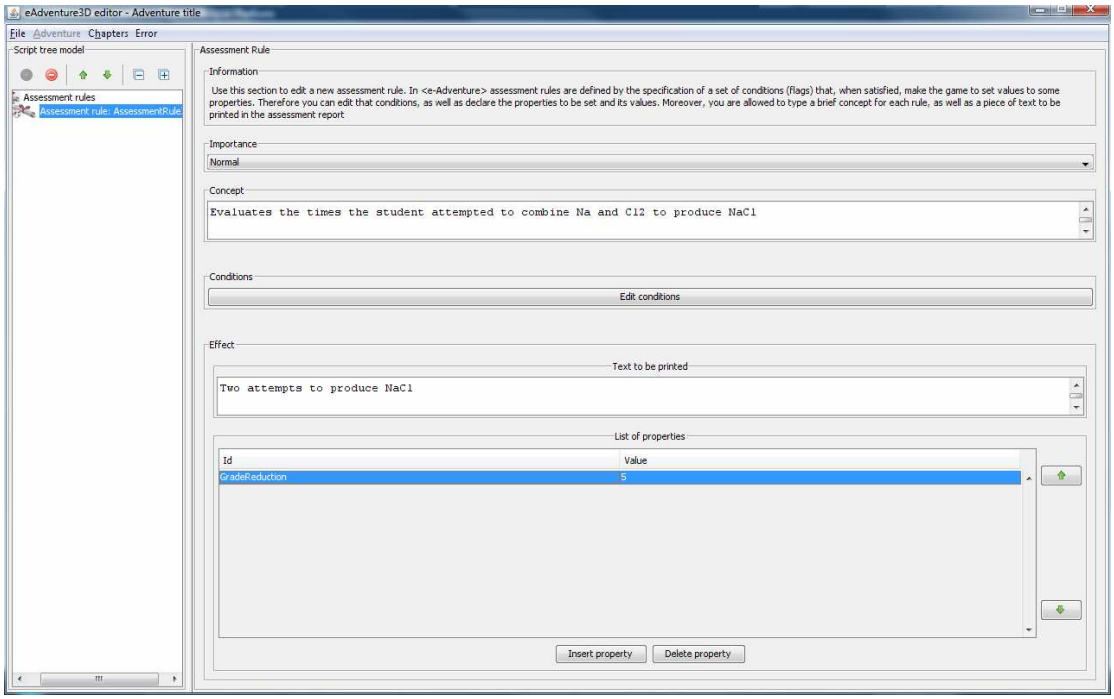
Uno de los puntos fuertes del proyecto es la posibilidad de definir reglas para la evaluación automática del estudiante. El instructor puede hacer uso de la supervisión que se puede realizar durante el transcurso del juego (gracias a la fuerte interacción jugador-juego) y producir automáticamente un informe con los resultados parciales o finales del estudiante. Además, podemos enviar este informe junto con un conjunto de variables a las que se les ha asignado un cierto valor a un LMS (Learning Management System o Sistema de Administración del Aprendizaje) para que esta evaluación sea añadida al perfil del estudiante.

El conjunto de reglas de evaluación automática (perfil de 'assessment') varía de un capítulo a otro. Cada perfil se escribe en un archivo XML diferente y se copia en el archivo EA3D. A continuación mostramos la DTD que define estos archivos XML:

```
<!ELEMENT assessment-rules (assessment-rule*)>
<!ELEMENT assessment-rule (concept, condition, effect)>
<!ATTLIST assessment-rule
   id ID #REQUIRED
   importance ( verylow | low | normal | high | veryhigh ) #REQUIRED
>

<!ELEMENT concept (#PCDATA)>

<!ENTITY % basic-condition "(active|inactive)">
<!ELEMENT condition (%basic-condition; | either)+>
<!ELEMENT active EMPTY>
<!ATTLIST active
   flag NMTOKEN #REQUIRED
>
<!ELEMENT inactive EMPTY>
<!ATTLIST inactive
   flag NMTOKEN #REQUIRED
>
<!ELEMENT either (%basic-condition;)+>

<!ELEMENT effect (set-text?, set-property*)>
<!ELEMENT set-text (#PCDATA)>
<!ELEMENT set-property EMPTY>
<!ATTLIST set-property
   id NMTOKEN #REQUIRED
   value NMTOKEN #REQUIRED
>
```

Las reglas de evaluación automática se definen fácilmente. Como se ve en la DTD, cada regla es definida por un grupo de condiciones (indicadores) y del efecto que se lanzará cuando se verifican esas condiciones. El efecto es determinado por un texto opcional que se imprime en el informe cuando se ejecute esa regla, y un conjunto de pares (variable, valor) que son las propiedades que se entregarán al LMS.

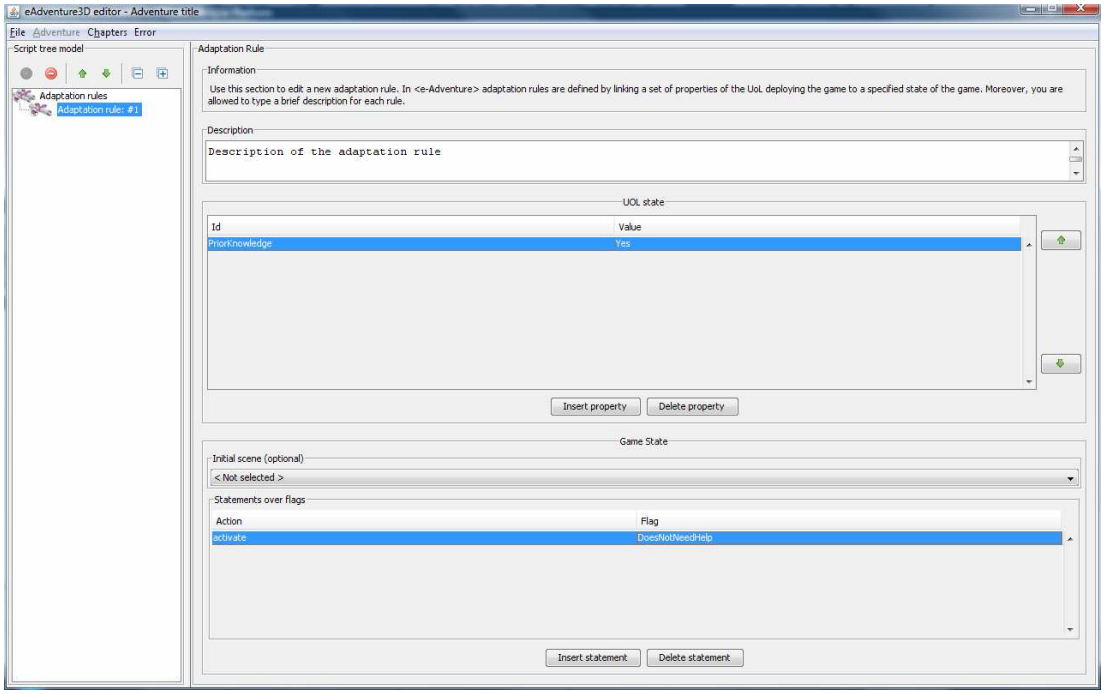
La siguiente figura muestra cómo es de fácil crear reglas de evaluación automática con el editor:



Adaptación

De una manera muy similar funciona la adaptación. Se pueden definir reglas de adaptación en las que se relaciona un estado de la Unidad de Aprendizaje (Unit of Learning o UoL), que es proporcionado desde el LMS como una lista de pares <atributo, valor>, con un estado del juego (representado en forma de activación y desactivación de indicadores). Además, se puede establecer una escena inicial distinta. Esto se puede utilizar, por ejemplo, para que algunos estudiantes avanzados salten algunos niveles básicos. Veamos la DTD a seguir para definir reglas de adaptación y un ejemplo de edición en el editor:

```
<!ELEMENT adaptation (initial-state?, adaptation-rule*)>
<!ELEMENT initial-state (initial-scene?, (activate | deactivate)*)>
<!ELEMENT adaptation-rule (description, uol-state, game-state)>
<!ELEMENT description (#PCDATA)>
<!ELEMENT uol-state (property*)>
<!ELEMENT property EMPTY>
<!ATTLIST property
   id NMTOKEN #REQUIRED
   value NMTOKEN #REQUIRED
>
<!ELEMENT game-state (initial-scene?, (activate | deactivate)*)>
<!ELEMENT initial-scene EMPTY>
<!ATTLIST initial-scene
   idTarget NMTOKEN #REQUIRED
>
<!ELEMENT activate EMPTY>
<!ATTLIST activate
   flag NMTOKEN #REQUIRED
>
<!ELEMENT deactivate EMPTY>
<!ATTLIST deactivate
   flag NMTOKEN #REQUIRED
>
```



De este modo los juegos de <e-Adventure3D> no son "cajas negras" como si lo son habitualmente los juegos comerciales: utilizamos la interacción entre el estudiante y el juego con el propósito de evaluarle y adaptar los juegos a sus conocimientos.

2.3.8 Conclusiones

Después de detallar todas estas características queda claro que existen muchas diferencias entre ambos proyectos (<e-Adventure3D> y <e-Adventure2D>). A primera vista podría parecer que definir una aventura en <e-Adventure3d> es un proceso más complejo que en la versión anterior. Sin embargo, en nuestra opinión hemos encontrado la manera de crear las aventuras gráficas en tres dimensiones fácilmente con resultados asombrosos.

Por una parte, si los usuarios desean escribir el guión de sus aventuras (que es totalmente innecesario porque con el editor se puede hacer todo gráficamente) encontrarían que es un trabajo sencillo a excepción del uso de cámaras (que a pesar de haber sido simplificado al máximo, aún necesita ciertos conocimientos de geometría básica) y las transformaciones. Sin embargo, si los fabricantes del juego tienen los modelos apropiados no tendrán que utilizar las transformaciones así que no tienen que saber nada sobre escalar o rotar modelos. Finalmente si utilizan entornos predefinidos descubrirán que es tan fácil como lo era en <e-Adventure2D>, ya que solamente tendrán que elegir una textura para las paredes de un cuarto cerrado o las imágenes del cielo en cuartos abiertos, en vez de una imagen de fondo como en la versión 2D.

Por otra parte, tenemos la herramienta de edición de juegos que permite que todo lo que se puede escribir a mano se haga gráficamente evitando los pequeños inconvenientes que no solventaba el lenguaje: no hay problemas con las cámaras ni las luces ni las transformaciones y, además, el proceso de creación de aventuras es más cómodo, ya que los recursos se almacenan automáticamente en el archivo EA3D. El hecho es que sin más esfuerzo que en <e-Adventure2D> se pueden desarrollar aventuras gráficas destinadas a la educación mucho más complejas y, por supuesto, mucho más realistas.

Como consideración final, queremos remarcar el alto valor educativo que pueden alcanzar las aventuras desarrolladas con nuestra plataforma. No sólo hemos incluido elementos dedicados especialmente a la educación, como las escenas de corte o los libros, sino que el resto de las características han sido pensadas para incrementar el valor educativo de los juegos. Las cámaras y las luces, por ejemplo, se pueden utilizar para promover la atención y la motivación, y para conseguir que el estudiante dirija su atención hacia los aspectos relevantes del dominio del estudio cuando sea apropiado. También los propios modelos en tres dimensiones con cualquier animación que el diseñador desee, permiten la obtención de un mayor realismo a la hora de enseñar detalles de ciertos aspectos que en dos dimensiones no era posible. Y por encima de estas características, la capa de evaluación automática y adaptación permite que los instructores calibren el comportamiento de los juegos según el estudiante y evaluar la experiencia de aprendizaje. Por otra parte, estos datos se pueden almacenar directamente en el perfil del estudiante en un servidor LMS, cuya cooperación es muy interesante si queremos que los juegos educativos formen parte de los planes de estudios y no sean tan solo material complementario que se utiliza de vez en cuando.

Chapter III

Domain Study. Motivation and Goals

1. e-Learning and educational videogames

During the last decades technology has evolved exponentially, especially in computer science. The speed of our processors and systems has increased exponentially, moving from the 66MHz of the Intel 80486DX2 processor (1992) to the last generation of multi-kernel processors, which clock speed (of each kernel) is in the order of GHz. Besides, Internet has involved a complete revolution in multiple areas such as business, communication, leisure time and entertainment, etc.

In short, the technology advances produced in the last decades (especially internet) have affected almost all the areas of our daily lives. That includes the learning area as well, a field in which technology has been applied since the 90s decade, in what has been named *e-Learning*. And the progress in this field has been surprising. Modern e-Learning systems have evolved from the simple repositories of mere static content they used to be in the early 90s to huge, complex systems of interactive content which cover all the aspects of the learning process for all kind of roles. Teachers are provided with features to assess, evaluate and guide the students during the process; content designers (role usually played by teachers as well) are allowed in those systems to create, store, manage and present digital contents stored in a centralized repository, etc. Those modern e-Learning systems are known as *Learning Management Systems* (LMS), which are in most of the cases web-based, since web contents are easy to design, store and above all, to deliver to the students in lots of situations and contexts, due to the broad acceptance of Internet nowadays [4].

Figure 1. Screenshot of the UCM (Universidad Complutense de Madrid) virtual campus, based on a WebCT LMS [40], one of the most extended.

There are numerous advantages of this web-based learning approach. Students are required to adopt a more active attitude in the learning process and develop further self-taught learning skills, as opposite to the passive process in which a teacher provides the knowledge to the students who are only expected to sit and listen. In addition teachers can use these online tools to dynamize the courses and to promote collaboration between students [16] (the use of forums and other communication tools is nowadays broadly extended and most of current LMS support them).

However, there are some shortcomings too. The disconnection between the integrants of the learning process hinders the supervision of the instructors, which prevents in some cases the achievement of the learning objectives. This barrier turns into a severe problem for those students which require, due to any reason, a closer guidance of an instructor. Besides, online courses have presented high drop-out rates ought to frustration and lack of motivation of the students [36, 37]. One alternative to overcome such disconnection is the *b-Learning* approach, in which traditional and online learning are mixed in the courses. Nevertheless those are still problems to be tackled.

On the other hand, video games have been pointed out as an interesting alternative in education due to some features that make them valuable for learning [3, 26, 28]. Psychologists and pedagogues agree that mammals (especially humans) learn while they play; in other words, the most natural way to learn is through play. It means that a satisfactory learning experience should be

entertaining for the learner (fun is a by-product of good learning). Besides, video games have become the most relevant industry of entertainment (now video games invoice more incomes that films) and their acceptance is not limited to children but extended to all kind of people in terms of age, race and economical situation. In contrast to the lack of attention that students usually present (the myth of the 10-minute attention span) video games are able to keep people concentrated during hours in the resolution of complex tasks which usually require reasoning and problem solving.

Another advantage is that modern video games produce an intense interaction between the player and the environment (i.e. domain of study), which is also a good source of information about the learning experience to be used to guide the students (is the student wandering around? Is the player lost?), to evaluate them (how many times have attempted to accomplish a task? how long did it take to solve a problem?), or to adapt the game depending on the condition of the student. Finally they can be used to promote collaboration between the students.

Taking all these advantages into account, video games seem to be a good innovation in e-Learning, a field which is being criticized due to the lack of motivation of the students [36]. Moreover, the discussion is not any more about whether video games can teach and provide benefits to the learning experience, but if those benefits are worth the effort and the cost. The main barrier for the introduction of videogames in educational contexts is their high development costs (according to Aldrich in [16 p210] 15 person-years and around 100,000$ and 500,000$ according to Michael & Chen in [41]), which are several orders of magnitude above traditional approaches.

Another disadvantage is that video games will not teach by themselves. Just enveloping the contents with an attractive game will not make the students learn them. To provide real learning experiences video games must be developed leveraging the educational and fun factors [19]. Besides they should be integrated with the rest of the curriculum and the performance of the students monitored to ensure that the learning objectives are being achieved.

So there is the discussion at the moment: if the learning experience provided by videogames is that better to afford their introduction in the educational system, since there are no evidences to support that. On the other hand, the problems of the high development costs, the support of mechanisms to monitor the performance of the students when they play and their introduction as another resource in the educational system are issues that need to be tackled. There is where the <e-Adventure3D> project tries to contribute. The <e-Adventure3D>

platform will attempt to provide a complete authoring environment for the development of educational videogames at a low cost, which can be easily introduced in online environments and which support the monitoring of the students.

2. Domain study: the three-dimension challenge

In the previous head it has been described the advantages that videogames can bring to education. In this section we carry out a brief study of the domain, including an analysis of the experiences with videogames in educational contexts (section 2.1) and which authoring tools we can find for videogame creation (section 2.2) as a study of the similar applications (related work) we can find in the market.

2.1. Experiences with videogames in education

Video games have already been successfully applied to education thanks to diverse initiatives. The most simple approach (but effective, however), has been just using Off-The-Self games (i.e. games developed for commercial uses which have just been taken from the market). Some of those games are rich enough to provide a good lesson.

That is the case of the strategy game *Civilization*, one of the best seller videogames in time. The reality of the historical scenarios reproduced in these games has been used in history lessons (although the historical facts occurred in the game could be inexact the student acquires an accurate perception of how the life was in older times). In [20] it is described an experience with students in a history course with Civilization III.

Figure 2. Screenshots of the Civilization III and Sim City games.

Another good example is the case of the *SimCity* saga. In these games players are entrusted to manage all the aspects of a city as if they were the major. The

success of the title facilitated the apparition of other games such as *SimFarm*, *SimHealth* or *SimEarth*, published by Maxis studios, which has been applied in educational contexts as it is described in [21, 22][3].

On the other hand, other videogames have been specially designed to cover the teaching of a specific domain. That is the case of *The Monkey Wrench Conspiracy*, published by *Think3,* which teaches the use of the industrial design software of the company. Another example is *Virtual Leader,* developed by *SimuLearn* for the teaching of complex aspect as "leadership" is. Finally, as a last example we can highlight the game *Objection!* developed by *TransMedia*, which has been certificated as a formation tool valid to get credits in law by the *Continuing Legal Education* (CLE) programme in USA[3].

Figure 3. Screenshots of *Virtual Leader* (left) and *Objection!* (right).

A case which deserves special mention is *Second Life ® I* [8], from *Linden Lab*. *Second Life* is a 3D virtual world which is created by its residents (people are represented into the Second Life world by an avatar they choose, which is used to interact with the world). In Second Life residents can carry out with their avatars multiple of the daily tasks in real life, such as customize their look (go to the hairdresser's, change cloths), produce digital content (which intellectual rights belong to the creator) or buy or sell their belongings (including their intellectual properties) using Linden dollars, which can be converted to US dollars. The interaction in the world is so rich that thousands of organizations such as IBM or politic parties have created their own virtual worlds in Second Life.

[3] Other experiences with commercial videogames are summarized in [1], from which these examples have been taken.

Figure 4. On-line lecture class of the Harvard Law School.

Motivated by the high acceptance of the Second Life simulation lots of colleges, universities and other educative organizations have decided to take advantage of its educative possibilities [7, 9] creating their own virtual campus where students can attend lecture lessons, participate in events with other students and teachers (promoting collaborative learning) or access the online educative contents. As it is described in [5, 6], Second Life is a good tool for the experimentation on diverse social issues with students, business training, or in general to the teaching of lecture sessions of all kind of subjects.

Although all these experiences have obtained satisfactory results, the applicability of this kind of games is very limited. Those are a closed product, which absence of flexibility hinders the reusability of the contents in other circumstances and forces the instructor to rely on pre and post-tests to check if the learning experience is being attained [39].

In addition, instructors cannot intervene directly on the development of these games; thereby the instructional design of the games weakens (or even in some cases is relegated to play a secondary role), as the people with the proper knowledge to devise it cannot take part on the implementation of the games. Moreover their high development costs are out of the budget for most of As a result these approaches do not seem to be the most appropriate for education as it is not viable to afford their high development costs if you cannot reuse them later (above all when educative contents are likely to change and vary and therefore instructors must be able to maintain the instructional design).

2.2. Authoring tools for commercial and educative videogames creation

As the previous section exposes, videogames are implemented as "black boxes", products which are difficult to maintain and adapt when their production is over, and which development costs are too high for educational budgets. That is an inconvenient for educational applications as learning content is expected to change in time (official curricula may vary, contents might be adapted for different courses or levels, etc.). Besides, instructors need to take an active role on the development process for educational videogames or the educational values will get lost (neither programmers nor game makers will know what students must learn).

Some of those shortcomings get solved (or at least palliated) by using authoring tools for the development of the games. Some are the initiatives which have caused impact of different considerations. Nevertheless, there is non authoring tool for videogames development covering all the countless features that modern games can be provided with. Consequently, the most representative examples have succeeded thanks to get expressivity and simplicity leveraged. Usually this is achieved by narrowing the application to a single game genre (first person shooters, sports, strategy, platforms, etc.). This is the case of some applications, spanning from free, simple initiatives to professional (and quite expensive) complete authoring platforms.

Between the most relevant we find *The FPS Creator* [11] (utility for the creation of first person shooter games), *The 3D Game Maker* [14] and *Game Maker* [29]. The first two examples allow users to create complex 3D games with just a few mouse clicks, while the last has been used as development tool in diverse projects of academic investigation, as it is described in [23, 24, and 25]. Other initiatives are *Alice* [30] and *Mission Maker* [31], tools which are devoted for education.

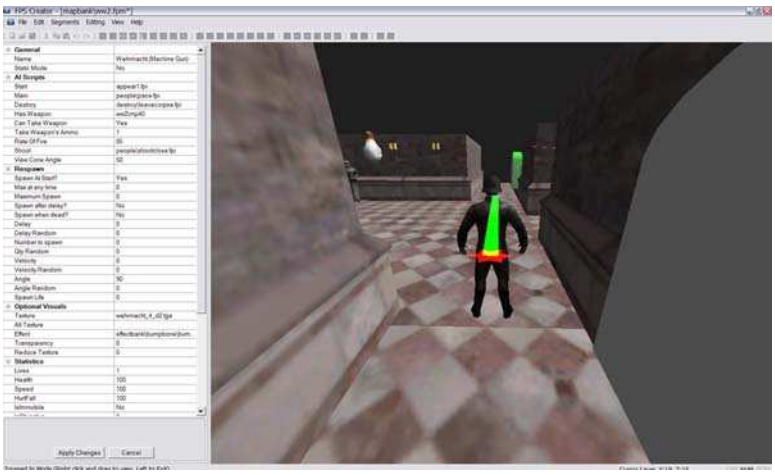


Figure 5. Screenshot of the FPS Creador tool.

Nonetheless, not all the game genres show the same features for education. As it is described in [1], almost all the genres present characteristics that can be used for educational purposes, but the most relevant in accord with some authors [18, 26, 27, 28] is the Adventure genre. When we talk about adventure games we refer to games such as *Myst*™ or *Monkey Island*™, in which the player must solve tasks through the exploration of the environment and the interaction with objects and other characters. The next section describes more thoroughly the features that make this genre especially suitable for education, but for now just highlight that in those games what is important is the narration (i.e. the contents to be transmitted).

Figure 6. Images of different adventure games of the Lucas Arts factory. The two on the first row are classic titles on 2D such as *The Curse of Monkey Island* ® and *Full Throttle* ®. The second row show the latest adventures which are represented in 3D (*Escape from Monkey Island* ® and *Grim Fandango* ®).

There are some adventure-maker tools available which allow authors without programming background to create their own adventure games. Between them *Adventure Game Studio* [32] and *Adventure Maker* [33] are perhaps the most popular.

Special mention deserves a platform of recent creation: the <e-Adventure> project [2] (<e-Adventure2D>), which last release (0.4b) is still a beta version, but

which effectiveness for the creation of educational *point-and-click* adventure games has been proved in several cases. Maybe this is the unique example of an authoring platform for adventure videogames which has been specially designed for instructors and with education-specific features.

Although these applications have been proved to be effective for the creation of adventure games, the games produced are limited to a 2D representation which differs to the current trend for modern commercial adventure games, usually created in 3D environments. The *3D Adventure Studio* [12] tool is promising, but its development seems to be halted at the moment.

3. General goals

As it has been exposed in the previous sections, videogames present interesting features that can be used for education. The discussion is not any more about what videogames can teach [3], but if it is worth the effort. The cases of study in which videogames have been applied to education are still few, although some obtained satisfactory results. Besides, the development costs of those games are in most of the cases excessive for educational budgets, and the creation and maintenance of the learning contents by instructors, who have the responsibility to guide the learning experience, are difficult when no authoring tools are involved in the process. Moreover, the efficacy of the learning experience cannot be checked by the instructors as no mechanisms are provided.

An approach to tackle all these considerations (which is the base of the work proposed in this document) is to develop authoring tools for the development of educational videogames without need of programming background. In this manner, instructors (who do not usually have such knowledge) can intervene directly in the production of the games, and later modify and maintain the learning contents, which in education are expected to need revisions very often. Moreover, the encapsulation of the programming tasks thanks to these tools is the key to reduce the development costs (the total development cost is reduced to the production of the art assets).

In addition, the most suitable game genre for education seems to be the adventure genre. Authoring tools for the development of these applications are available, but the most relevant examples are 2D game makers, which are not in accord with current game development trends. It is true that educational videogames do not need to use the last generation technologies as their objectives differ from the commercial game industry, but it is very recommendable that those are as close to what students expect from a videogame as possible. If current

adventure games are developed as 3D videogames (which is a consequence of what the public demands) that is what a student will expect from an adventure videogame. Moreover, the interaction and realism obtained in 3D adventure games is much richer than in 2D worlds since the navigation in the first is closer to the reality (there are things that cannot be represented in a two dimensions world, or at least will be better represented in a 3D world).

The above paragraphs set the main objective of the project:

(1) *Produce an environment authoring platform for the creation and execution of 3D educational adventure videogames with no need of programming or game-making skills with a low production cost.*

As it has been exposed, there are some good authoring environments for videogames development, but focused on commercial videogames which differ from educational videogames, which are expected to show other, education-specific features. Between these the most relevant are the ability to auto evaluate the students (assessment), to adapt the games according to the profile of the students (adaptation), and the capability of integrate the games in current Learning Management Systems so the results can be attached to the profile of the student.

As a consequence, the second objective of the project is the following:

(2) *Endow the platform with features for the assessment and adaptation of the learning experience. The platform must present the ability to communicate with Learning Management Systems so the evaluation produced can be attached to the profile of the student and the adaptation is carried out according to it.*

As it has been exposed, the aim of the project is to tackle the high development costs of educational videogames by the reduction of the cost involved in the programming and game implementation tasks. It is not the intention of this project to reduce the costs of the art assets production. There is little that we can contribute there as there are plenty of authoring tools for the creation of images, sounds, videos or 3D models.

Finally, there is an underlying objective that we would like to point out. The experiences with 2D point-and-click makers have obtained good results. Narrowing the scope of these tools to the adventure genre reduces the development costs as the common elements of the genre can be extrapolated easily. On the other hand, the production of 2D games is reasonably simple from a technical point of view since games are compounded just by images which are rendered one over the other on the display system. However, there is no evidence a priori that the same reasoning will be valid for the production of 3D games. In this case the technical complexity goes beyond the line that is "under our control". Besides, the expressivity of 3D games makes the process of getting the common elements extrapolated much more complex since expressive power and development simplicity must be leveraged carefully. That is the key for the future success of the project.

4. Description of the work

The previous section has established which the main goals of our project are. In this section we will give a brief scope of how the project will be developed and implemented to achieve the goals proposed.

As objective (1) describes, we will produce a platform for the creation of 3D adventure games for educational purposes. A priori, we will follow the procedure used in the development of <e-Adventure>. The <e-Adventure3D> platform will be composed of two applications: a game engine which will run the games, and a game editor for the creation of the games.

The main part of the project is to develop the game engine, along with the language used to implement the games. Following the ideas exposed in [1], we will take a documental approach for our games. Thus the games will be written in plain text, human-readable documents that the engine will interpret and execute in combination with the art assets of the game, which are left aside (the storyboard of the games is split from its representation). Thereby instructors can write the executable storyboards with the help of just a text editor, which reduces both the development costs of the games production and the costs of developing an authoring tool.

For that purpose we need to design a game language, based on XML technologies, which leverages the expressivity (what can be expressed with the language) and simplicity. On the one hand, if the language is not expressive enough the games produced will be very similar and probably will not be able to "teach" anything. On the other, if the language is too complex instructors could not

write their own games using the language.

However, it is likely that setting a 3D game with just a text editor will be too tricky for instructors. The adventure genre does not cover problems such as artificial intelligence design, complex particle effects, etc. But, a priori, instructors will need help to set the items and characters in the games (3D positioning skills are required, and the elements could need to be scaled or rotated which are not easy tasks), or to create complex conversations with hundreds of dialog lines. For those tricky tasks we will develop an editor tool, which will help instructors to develop the adventure games.

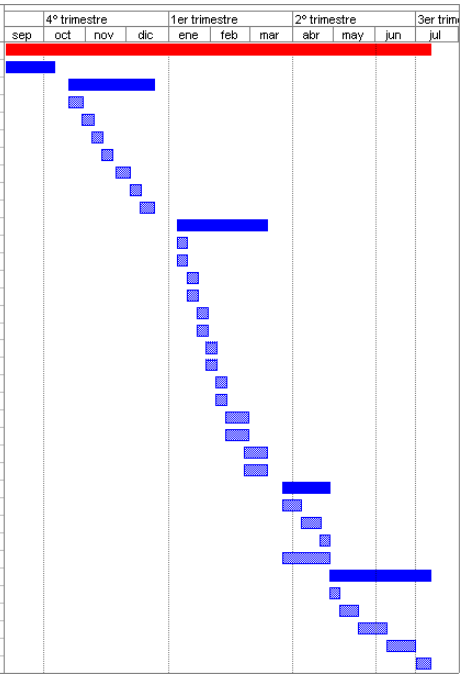
Chapter IV

Project Management

1. Project planning

The project planning is divided in five iterations. The first one is a preliminary iteration which its main goal is to choose the technologies and develop a technological prototype to prove the potential of these technologies. After this iteration there are planed three main iterations which their objectives are to develop the engine, develop the editor tool and the connection with the LMS server, respectively. The final iteration is concerned for improvements and extensions of the project. Iterations are divided in micro iterations of one, two or three weeks in order to control that everything is going as it is planed. The detailed tasks of these micro iterations have been planned at the beginning of the corresponding iteration. The next picture shows the Gantt diagram of the project planning:

Id	Nombre de tarea	Comienzo	Fin
1	**e-Adventure3D project**	**lun 03/09/07**	**vie 11/07/08**
2	Preliminary iteration: technological prototype	lun 03/09/07	lun 08/10/07
3	**First It: engine development**	**vie 19/10/07**	**vie 21/12/07**
4	Proyect design and DTD definition	vie 19/10/07	lun 29/10/07
5	Load models from an XML file	lun 29/10/07	mar 06/11/07
6	Input management	lun 05/11/07	mar 13/11/07
7	Hud development and colision detection	lun 12/11/07	mar 20/11/07
8	Objects interaction, game state definition and inventory	vie 23/11/07	lun 03/12/07
9	Conversations, javadoc and adventure test	lun 03/12/07	mar 11/12/07
10	Documentation	lun 10/12/07	vie 21/12/07
11	**Second it: editor development and engine extension**	**lun 07/01/08**	**jue 13/03/08**
12	Editor: design	lun 07/01/08	lun 14/01/08
13	Engine: unresolved tasks from first iteration	lun 07/01/08	lun 14/01/08
14	Editor:first steps with scenes edition	lun 14/01/08	mar 22/01/08
15	Engine: sounds in conversations	lun 14/01/08	mar 22/01/08
16	Editor: complete scenes edition	lun 21/01/08	mar 29/01/08
17	Engine: game menu (load and save)	lun 21/01/08	mar 29/01/08
18	Editor: characters and items edition	lun 28/01/08	mar 05/02/08
19	Engine: cutscenes	lun 28/01/08	mar 05/02/08
20	Editor: conversations edition	lun 04/02/08	mar 12/02/08
21	Engine: books	lun 04/02/08	mar 12/02/08
22	Editor: conditions and effects edition	lun 11/02/08	jue 28/02/08
23	Engine: improve effects system	lun 11/02/08	jue 28/02/08
24	Editor: documentation	lun 25/02/08	jue 13/03/08
25	Engine: documentation	lun 25/02/08	jue 13/03/08
26	**Third it.: LMS server connection and extensions**	**lun 24/03/08**	**lun 28/04/08**
27	Cutscenes and books in the editor + Lights in both	lun 24/03/08	lun 07/04/08
28	Regions + Move character + Chapters + Model rooms	lun 07/04/08	lun 21/04/08
29	Add new effects + background sounds in scenes	lun 21/04/08	lun 28/04/08
30	LMS server connection: Adaptation and assessment	lun 24/03/08	lun 28/04/08
31	**Final iteration: Revisions and improvements**	**lun 28/04/08**	**vie 11/07/08**
32	Tech demo + Documentation	lun 28/04/08	lun 05/05/08
33	Extensions	lun 05/05/08	lun 19/05/08
34	Tests + javadoc + Appearance improvements	lun 19/05/08	lun 09/06/08
35	Develop the final report	lun 09/06/08	lun 30/06/08
36	Prepare presentation	mar 01/07/08	vie 11/07/08



2. Risks management

We are going to manage the possible risks of the project based on the ideas of the book “Software Engineering” written by Ian Sommerville. We manage risks in four steps:

First step: Risks identification

The goal of this step is to identify the possible risks of different types:

1. Technology risks:
    - Not implemented functionalities in JME
    - Possible problems with JME inside swing components.
2. People risks:
    - Required training for staff: development team formation in 3d technologies (JME).
    - Time dedication of the components in periods such as holidays and exams.
3. Estimation risks:
    - The size of the software is underestimated.
    - The time required repairing is underestimated.
    - Super estimate the reusable code from <e-Adventure2D>.
4. Resources risks: Problems finding assets to use in test games such as models or textures.

Second step: Risks analysis

After identifying the project risks we are going to analyze them in a table. We are going to measure the probability each risk has of happening and its possible consequences. This probability can be very low, low, moderate, high and very high. As well as we do with the probability, we do the same with the effects these risks can cause, the effects are classified in catastrophic, serious, tolerable or insignificant. We can see this information in the next table:

Type	Risk	Probability	Effects
Technlogy	Functionalities not implemented in JME	High	Catastrophic
	Problems with use of JME inside Java swing components	Moderate	Serious
People	Development team formation in java 3-d technologies	High	Tolerable
	Time dedication of developers during exams and holidays.	High	Tolerable
Estimation	Underestimated the size of the software	Moderate	Serious
	Super estimate the reusable code from <e-Adventure2D>	Moderate	Serious
	Underestimated the time required for repairing	Low	Catastrophic
Resources	Problems finding assets to use in test games such as models or textures.	High	Tolerable

Third step: Risk planning

Some risks like the dedication time of the developers in holiday periods can be prevented if these periods are considered in the project planning. Other risks such as the third risk of the table can be minimised by forming the development team in this new technology in previous phases of the project. However, there are some risks that can be catastrophic for the completion of the project. In this case the second risk could determinate the potential of the editor, so we must consider alternative plans in case the risk happens. This table shows the strategy plan for each risk:

Type	Risk	Strategy
Technlogy	Functionalities not implemented in JME	Develop a technological prototype in order to make sure that JME is capable of doing what it is supposed to do. However, if a requirement can not be achieved we should reserve enough time of the project planning in order to develop it.
	Problems with use of JME inside Java swing components	If there are a lot of problems including JME windows in the editor tool panels we must take a contingency plan: we can use previews windows outside the swing components
People	Development team formation in java 3-d technologies	Start the project a few months before October in order to develop a technological prototype to learn the basics of Java 3d technologies (JME). It will minimize this potential of this risk.
	Time dedication of developers during exams and holidays.	Bear in mind this periods while stabilising the project planning in order to avoid its consequences.
Estimation	Underestimated the size of the software	Be careful with the extensions we want to develop in the final iteration. If we find out that an improvement requires much more time that we have expected we should cancel it.
	Super estimate the reusable code from <e-Adventure2D>	If code from <e-Adevnture2D> is not easy to be reused, we should develop our own code because at last term it will be more productive.
	Underestimated the time required for repairing	Time for repairing the project at the planning should be enough in order to finish the project on time.
Resources	Problems finding assets to use in test games such as models or textures.	Buy some models and textures, hire someone that can design them or form the development.

Fourth step: Risk monitoring.

The goal of this step is to monitor identified risks. Every week the project director establishes a day to monitor project development process through a meeting. We use this weekly meeting to decide whether or not a risk is more probable of happening and to apply the strategies of the third step when they are necessary. There are some clues that can advise us about a risk that is about to happen:

Type	Potential indicators
Technlogy	Neither the documentation nor the JME forum indicates that something we need to use in the project is included in JME.
People	The development team has not worked before with this kind of technologies. Or the planning time is being compromised because holidays
Estimation	Milestones of the week are not being reached as we can see in weekly meetings when we compare with the project planning.
Resources	We can not find free resources in the Internet.

Chapter V

Requirements Analysis

1. Requirements

In this section we are going to explain the project requirements. There is a distinction between the editor and the engine requirements. We also differentiate between functional requirements and non-functional requirements.

Functional requirements define what the system should do; in this case, our functional requirements are mainly about the final product we want to obtain. These requirements where completely defined by our project director at the beginning of the project.

Non-functional requirements speak about the conditions under which ones the project can work. Mainly these conditions are determinated by the 3D engine used and the necessity of an internet connection. He divided non-functional requirements in two categories: software and hardware.

2. Engine requirements

2.1. Functional requirements

1. Run 3d educational games: The engine should allow students to play 3d educational games based in graphical adventures and written in <e-Adventure3D> language (XML).

2. The engine should be independent of any platform and easy to be installed: The project is going to be used in different places and situations by different people. Moreover, it is supposed to be run in the Internet so it must be a multiplatform system. For the same reasons it must be easy to be installed.

3. Allow a big range of file types: The engine should allow the use of different types of 3d model files. The system does not have to allow the creation of models, but it should allow the use of different file types or it will limit its functionality a lot. So that platform must allow the use of models that can be created with different programs. For the same reason, it also must allow different types of images files, audio files and video files for the same reason.

4. LMS connection: The engine must be capable of making a connection with a LMS server in order to take the most of games as an educational source. The LMS server allows the system to interchange information in order to adapt the games to students' level and knowledge while they are playing the game. It also must

generate reports of the student evolution.

5. Make easy the extension of the project: Looking to the future, project code must be well structured and documented in order to allow the extension and maintenance of the project.

2.2. Non-functional requirements

Software

1. Java version 1.6.0_02: the project code is compiled in this java version.

2. Internet navigator: In order to run the project from the internet as a java applet.

Hardware

1. Graphics card: In order to play 3d games a graphics card that supports openGL is needed. The specific requirements of the graphics card depend on the 3d models used in the games developed.

2. Minimum screen resolution of 1024X768.

3. Internet connection or connection to a local area net: This requirement is only necessary if the adventure to be played uses a LMS connection. The type of connection depends on the place where the server is connected. If the server is in our own net, a local area net connection covers it; however, if the server is not in the same net as the player, then an internet connection will be needed.

3. Editor requirements

3.1. Functional requirements

1. Create 3d educational games in a graphical way: The editor tool should allow teachers to create their own educational games without knowledge of XML.

2. An intuitive interface: the editor tool is supposed to be used by teachers. They could not be experts in computers so the editor tool must have an intuitive and friendly interface in order to minimize the learning process and abstract the difficulties.

3. The editor tool should be independent of any platform and easy to be installed: This requirement is shared with the engine.

4. Allow a big range of file types: The same file types required by the engine.

5. Make easy the extension of the project.

3.2. Non-functional requirements

Software

1. Java version 1.6.0_02: the java code is compiled with the same java version of the engine.

Hardware

1. Graphics card: In order to show preview windows that contain 3D models in the editor a graphics card that support openGL is needed. The specific requirements of the graphics card depend on the 3d models used.

2. Minimum screen resolution of 1024X768.

Chapter VI

Use Cases Specification

1. Use cases

In this section we are going to capture the functional requirements of the previous section through the use cases. We are not going to give a detailed description for the behaviour expected from the system for each use case. As always, this chapter is divided in use cases for the engine and use cases for the editor tool. This section covers the actors involved in the use cases and the list of use cases for each actor.

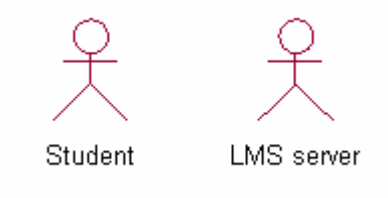
Actors characterise a role of persons or external systems that interact with our system. As we have said many times, this project was conceived to create educational games. The use process can be detailed as follows: *Teachers* are in charge of creating the games so the *students* can play these educational games developed by the teachers and while they are playing, system can connect with a *LMS server* in order to adapt game difficulty and generate informs. As a result, three actors take part in <e-Adventure3D>: teachers, students and the LMS server.

Actors are linked to one or more system use cases and use cases are procedures that can be made by an actor.

2. Engine

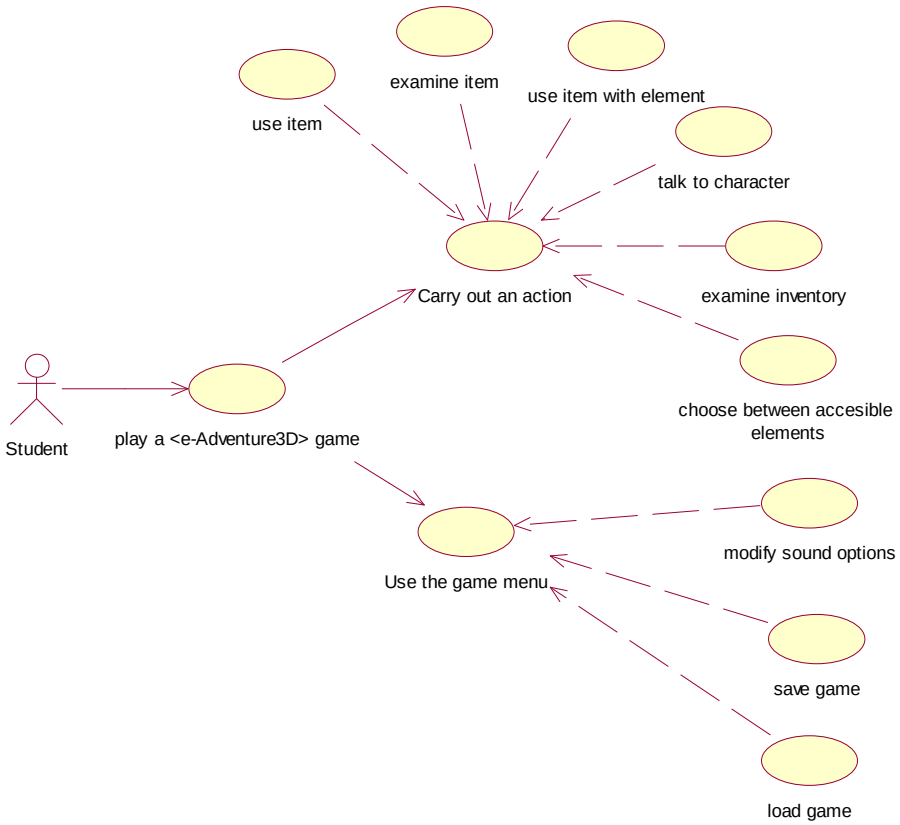
2.1. Actors:

Two of the three actors mentioned can take part in the engine: the students and the LMS server:



2.2. Use cases:

We are going to see the basic use cases linked to the students:



The student plays the games developed by the teacher. Play a game involves interacting with the elements of the scene. The student can change some game options through the menu.

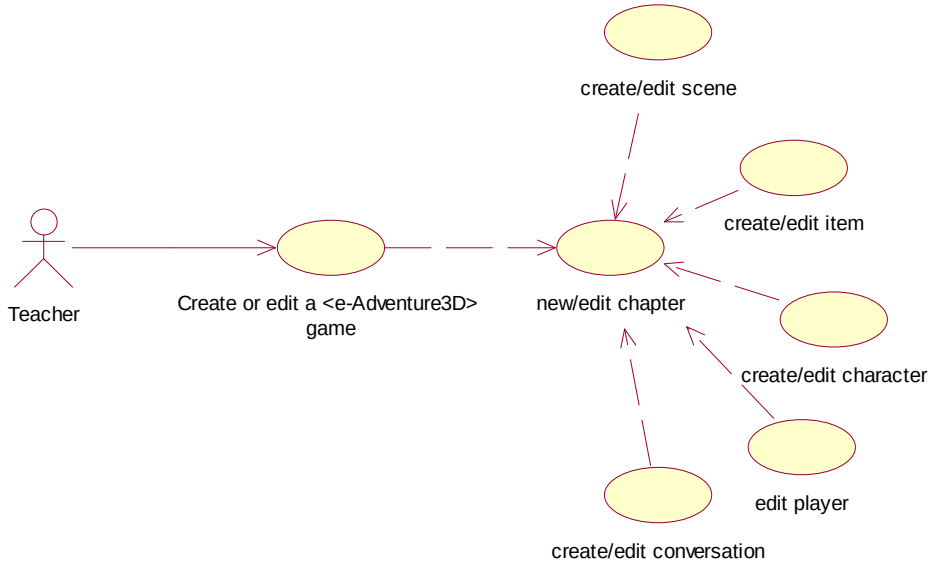
3. Editor tool:

3.1. Actors:

The other actor mentioned in this chapter introduction is the teacher. He is the actor that uses the editor tool. Perhaps there should be another actor called art designer which would be the one in charge of creating the art assets.

3.2. Use cases:

The main use case for the teacher is to create or edit educational games. This use case involves creating scenes, items, characters, conversations, etc.


create/edit scene
create/edit item
Teacher
Create or edit a <e-Adventure3D> game
new/edit chapter
create/edit character
edit player
create/edit conversation

Chapter VII

Design

1. Design

This chapter explains the design of the engine and the editor tool. Both are part of the same project so despite of being completely independent all the project code is into the same package called *es.eucm.eadventure3d*. This name was chosen because the web page of the university department (http://www.e-ucm.es). It is a common practice to invert the internet address and use it as the name of the package of the project in order to be unique (as unique is the URL). During this chapter we will omit this part of the name when we speak about packages.

Firstly, we explain the packages that contain the common code shared by the engine and the editor tool. Secondly, we explain the code design for the engine and, at last, we explain the editor tool code.

2. Code shared by the engine and the editor tool

The whole project (the engine and the editor tool) is divided in three packages: *engine*, *editor* and *common*. The idea is that the classes stored in the package *engine* have references to classes in the packages *engine* or *common*, the classes of the package *editor* have only references to classes of the packages *editor* or *common* and the classes stored in the package *common* only references classes of this package. This situation guarantees that the editor tool and the engine are independent one of the other, so that an executable JAR file can be generated for each one. In some cases, the use of this shared package turns difficult to see that we follow the pattern model-view-controller. However, this package was essential in order to avoid developing the same code in two different places which would has caused a higher cost of time and would has induced to errors.

There are three common packages shared by the editor tool and the engine. We are going to speak about each of them:

- *common.gamedata*: This package represents the part called model of the patter model-view-controller. There are classes that define the data required for chapters, scenes, elements, conversations, cut scenes, transformations, assessment, adaptation, etc. These are the data-container classes where the data parsed from the storyboard file and assessment and adaptation profiles reside. Therefore, these classes are independent of the 3DEngine chosen (moreover, they are virtually independent even of the 2D or 3D approach), and reflect exactly what can be specified in a storyboard file. As the storyboard files and the

assessment and adaptation files are the same for the engine and for the editor tool, these classes are also the same, so that they are stored in this common package.

- *common.loader*: Its responsibility is to provide a simple interface to load the data from the XML documents. So that, the function of this package is to parse the storyboard of the game and the assessment and adaptation profiles in order to store everything into the objects of the classes created in the package *common.gamedata*
- *common.url*: This package contains customized Java URL classes from ZIP files. The main feature is that the method openConnection() will return an InputStream from the zip.

3. Engine

3.1. Rules and guidelines.

The design of the engine has been thoroughly taken. It is a very important point of the work, as it conditions not only the implementation process but also the subsequent updates that could be performed some day. Thus the design process has followed some rules and guidelines, short and few but necessary, which will be next described.

3.1.1. The MVC pattern

All the work done has been directed by the Model-View-Controller pattern. So it means that data has been completely separated from an entity called “Controller”, which rules all the actions executed in the program and the course of the game. It also gives instructions to the “View Entity”, which is entrusted to show what the user has to see on the screen. That is the theory, but its application to <e-Adventure3d> is not so simple, since some external units had to be isolated (see “Isolation of the 3-D engine”) and the main loop of the game is directly controlled by the 3-D engine. In the following lines it is explained which entities in the project are matched to the Model, Controller and View, respectively.

Controller

There is not a single unit in <e-Adventure3D> which wholly represents the

"Controller Entity" on its own, as it has been divided in two different entities. The first one, located in the package *engine.core.control*, controls the evolution of the state and the story thread of the game. Therefore it is the main controller, as it is "who" takes the decisions of what has to be done in each moment, and how the "state" of the game is updated. It also uses a package, *engine.core.control.gamestate*, to delegate the actions that should be performed in each *main loop*[4]. Thanks to that, it is easy to identify different states of the game, and consequently to execute the suitable actions for each one. Currently some game states are distinguished, like *PLAYING, RUNNING_EFFECTS or CONVERSATION*. Further details are shown in section 1.2.4.3.

Secondly, the other "half" of the controller unit is located in *engine.core.gui.guicontrol*. This entity directs and coordinates all the logic which involves directly the use of the JME. That not only includes the output rendering, but also the movement of the player, the detection and resolution of the collisions with objects and characters, sound playing, cameras handling, etc. The input handling is treated in a different way, and deserves a special area to be explained.

JMEController is not a simple "GUI controller" properly talking, as it oversees areas that do not influence the course of the game but which are indispensable for the correct development of a 3D game, and imply a large load of programming work (for instance, the movement system of the player).

That separation was necessary. Looking back on <e-Adventure2D>, the Game class, helped by *GameState*, could control the whole game on its own, but passing from 2D to a 3D environment that approach would entail an excessive complication of the control unit. Moreover, it should be taken into account that the <e-Adventure3D> project has a complete dependence on the 3D engine chosen. This divided approach establishes a clearly-defined barrier between the <e-Adventure3D> core and the 3D engine, so after a reengineering process it could be replaced or improved by other engine.

Model

<e-Adventure3D> stores data in several ways. It needs to store meta-data, such as the movement settings (explain movement settings/reference), the input settings (reference to input settings), but the model, according to the MVC pattern,

[4] Following the most common design for digital games, the execution flows in a loop that ends when certain conditions are satisfied (i.e. the player wins or looses). For each iteration of the loop the game is updated according to the input state, and subsequently the image to be displayed is rendered.

is the one that comes from the XML storyboard file. That data is parsed when the file is read and the package which contains from that moment the data is stored in *common.gamedata*, as we explained in the previous section of this chapter.

View

Obviously, the view entity in <e-Adventure3D> is that which "talks" to the output rendering system (that is JME). It is located in *engine.core.gui*, and covers the scene rendering, HUD, Inventory and Conversations display (references).

3.1.2. Isolation of the 3D engine

As it was mentioned in the previous section, a 3D-based approach entails a great dependency with the 3D engine chosen. In our case, <e-Adventure3d> completely depends on JME. Although JME was chosen as our 3D engine, this does not imply it is the best java-based 3D engine available now and ever. It might be required, or desirable, to replace it by other for performance reasons, so a good isolation of the 3D Engine arose as indispensable. For such purpose different interfaces have been put between the <e-Adventure3D> engine and JME, in those places where both interact.

A special emphasis had been placed on the encapsulation of the input, which will be explained in the following lines.

In <e-Adventure3D>, there are several sources of interaction with the game. Those are:

- Player movement.

- Interaction with objects and characters.

- *List shifts* (for cases in which the user is shown some options to choose, like in conversations or when displaying the inventory)

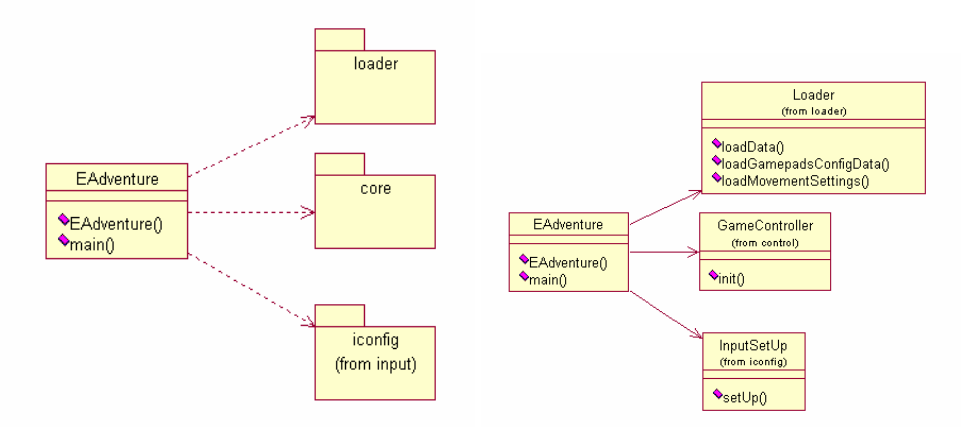
- *Debug interaction*. Obviously, it will be available only during the development stage.

For each "source of interaction" a handler has been defined, creating a layer between the Input System and the control. Each one is related to a controller unit, which knows "what to do" for every input signal. That "control layer" has been specified by interfaces, so then the input layer gets completely separated.

3.2. The design package by package

3.2.1. The class EAdventure3D

EAdventure3D is the very first class involved in any execution of the application. Its responsibilities are to launch the input set up process (see input design), load the game data and the movement settings from XML files (see loader), and finally launch the execution of the game (see *GameController*).



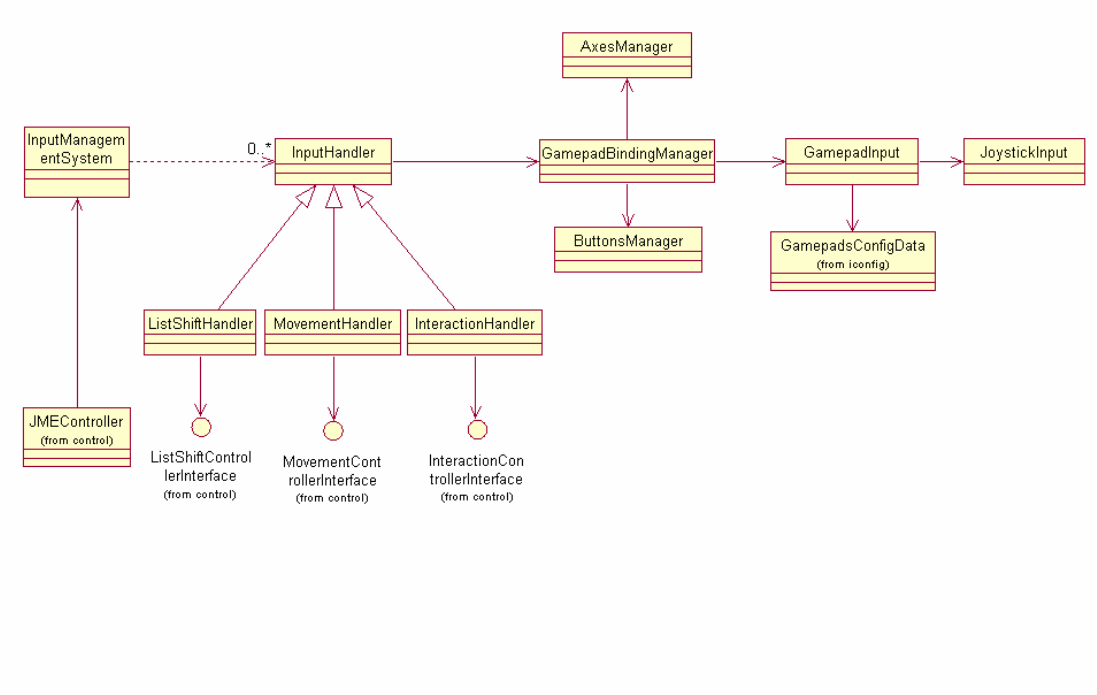
3.2.2. Package Loader

Its responsibility is to provide a simple interface to load some data handled by the <e-Adventure3D> engine that we call the property files. Those do not represent data concerning the game, but metadata. Specifically talking, <e-Adventure3D> manages two “properties files”, loaded and stored using XML representation: the movement settings file and the game pads configuration file. The first one stores properties about the movement. The second maps the input data provided by the JME layer to the <e-Adventure3D> representation.

3.2.3. Package Input

One of the most important points of the design was the input system. The <e-Adventure3D> needs were not covered by the low-level system provided by JME, so a lot of work had to be done here.

The proposed design is organized in 4 layers (including an external layer, provided by JME, called *JoystickInput*). The operation of each layer is further detailed in the implementation section, so here we will just give a scope of the whole system.



The first layer, implemented in the JME distribution, provides a polling-based system, consisted of two interfaces. One returns whether a button of the game pad is pressed or not in the moment of the poll, and the second gives the value of an axis (as a float value, 0 when not moved).

The second layer, *GamepadInput*, gets those values, and maps them to the "<e-Adventure3D> game pad". Thanks to the configuration values, previously obtained, *GamepadInput* maps each button and axis to one of the 4 axes and 13 buttons of the <e-Adventure3D>. Therefore, the layers below do not need to know anything about how the game pads are configured.

GamepadInput is used by the *GamepadBindingManager* system, which is the third layer. That system provides a higher level of abstraction, allowing the subsequent layers to "bind" a text string to a given sequence of input events. Thus, it is feasible to refer to a sequence of input events by a name, so it is easier to check if it is active in the moment of the update. Thanks to that complicated combinations of several buttons and/or axes can be defined. Moreover, it allows the simulation of analogue devices by either keyboard keys or game pad buttons.

The last layer, which is directly used by the *"control layer"*, is formed by a group of Handlers, each one designed to cover a specific way to interact with the application, and a "handler factory", the *InputManagementSystem*. *JMEController* creates the proper controllers for each interaction type, pass them to *InputManagementSystem*, and then it creates a suitable *InputHandler* that will invoke the controller attached when a valid input event is generated.

In this package is also contained the meta-data, such as the *gamepaddata* (used to store the game pad settings).

3.2.4. Package core

Once a solid framework has been established, providing accurate and *not-game pad dependent* input events (*input.ihandler* package) and tools to manage the XML data files (package common.loader), we can focus on the "heart" of the application, the package core. It is divided in three sub-packages following the MVC pattern (see section 1.1.1), namely *core.data, core.gui, core.control*. Those three will be next explained.

3.2.4.1. Package core.data

The package *core.data* contains the package *functionaldata*. It transforms the data loaded from the XML files, which contain the chapters of the game, to a real functional data. It means that *functionaldata* contains the data needed to operate in the game, adapted to the specific 3DEngine chosen. Consequently both Game and GUI controllers usually operate with functional items instead of the items themselves.

3.2.4.2. Package core.gui

This package uses the 3d engine to render the scenes through the class *GUIDisplay* that extends the class *'FixedLogicatreGame'* of the JME platform. There are a lot of other classes that can be extended to create games with JME such as SimpleGame or BasicGame. However, 'FixedLogicatreGame' is our best option because is a class that allows to take enough control of JME. The parts of the game to be render depends on the current state of the game and these parts are updated by the class JMEController that we explain in the next section of the chapter.

This package also contains some parts that must be designed by us, such as the HUD, the inventory, the game menu or the loading screens.

3.2.4.3. Package core.control

There is where the control layer resides. The structure the control package has been given is worth to explain, as three different entities take part (broadly talking).

Firstly, the main controller: **GameController**. This class controls the execution of the game, and therefore the way its state is updated. But before talking about state management, we have to be sure of what is a "state" in <e-Adventure3D>. The **state of the game** in a specific moment comes from:

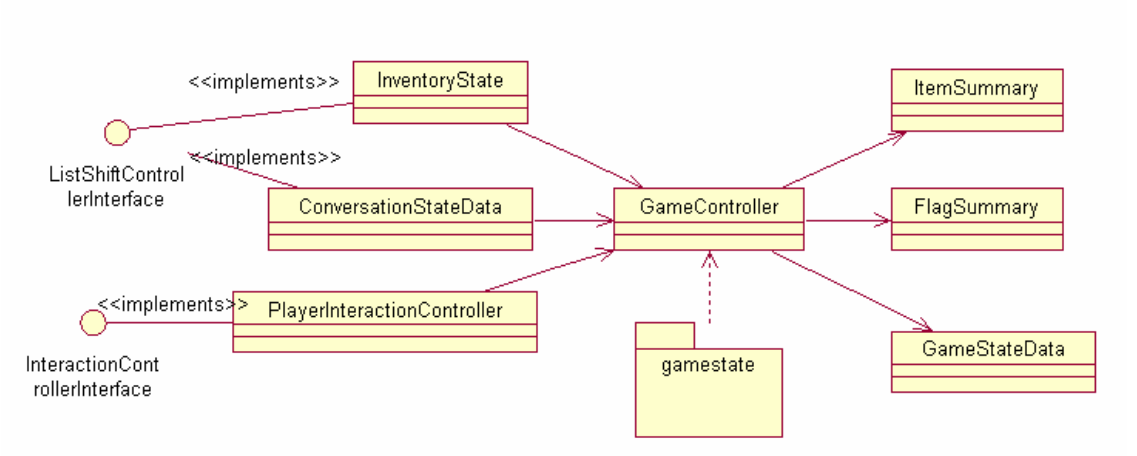
- The *state of the flags* (active/inactive),

- The *current scene* where the player is,

- The *player position* (as it determines which objects can be interacted in each moment)

- The *inventory* (the set of objects grabbed and not consumed by the player)

So to manage and update that "state" above described, *GameController* delegates to several classes (*ItemSummary*, *FlagSummary* and *GameStateData*), as it is illustrated by the figure below. Note that both *InventoryState* and *ConversationStateData* implements *ListShiftControllerInterface*. That is because those classes need to receive input signals when users have to decide with which object of the inventory want to interact, or what option in a conversation want to select. On the other hand, there is *PlayerInteractionController*, which implements *InteractionControllerInterface* for the same reasons. Both three classes use *GameController* to access and modify the state.

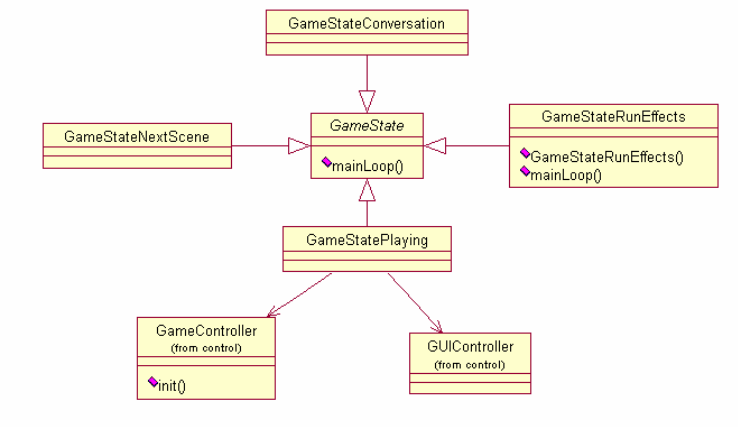


There is other entity involved there that is to be mentioned aside because of its importance: the **gamestate package**. It should not be confused with the idea of a "state", thought as a set of data in a specific moment. The *gamestate* package does a different consideration: it encapsulates the code that has to be executed in each different stage of the game. To clarify what is called *"stage"*, let's show an example: loading the scene vs. playing. When a scene is playing, the engine should show a loading screen, with a piece of text like "Loading…", but when playing it has to execute more, and more complex tasks. That is what we mean as "stage", the set of

actions that have to be executed in each moment.

The "piece of logic" that has to get executed in each update process is encapsulated in the method *mainloop()* of each *GameState* class. That method invokes the operations that *GameController* and *JMEController* have to execute in each case.

Let's see a class diagram with some of the most useful stages:



Thus far, some different stages have been defined and implemented in <e-Adventure3D>

- *Playing*. In this stage, the movement of the player is enabled, and therefore collision detection and camera handling (avoid getting the camera out of the scene, changes of camera, etc.) are needed. It is also necessary to render the scene, detect the set of objects which are available for interaction, and support such interaction in the event of the associated button is pressed. This is the state used during conventional scenes.
- *Running Effects*. The running effects stage is launched when the player interacts with an object or character. During this stage, every effect specified in the proper section of the storyboard file gets executed. For each different effect the suitable actions are taken. Once there are no more effects to execute we go back to the Playing stage.
- *Run effects during conversation*. This stage is used during conversations because any effect can be launched in each line of a conversation. It does the same actions we specified in the stage running effects but here we go back to the conversation stage.
- *Video*. This stage is used during video scenes or when a video is launched as an effect. It plays the video until it ends and then returns to the stage of

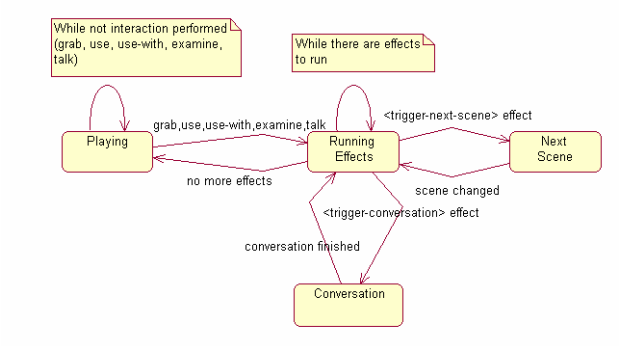
running effects[5] or to the stage next scene.

- *Slide*. This stage is used during slide scenes or when a slides effect is launched. It shows all the slides of the scene one after the other. When there are no more slides it changes to the stage running effects or to the stage next scene.
- *Book*. This stage renders the book in the screen. When users have read the book it goes back to the running effects stage.
- Load data. This stage loads a saved game. When the game is loaded we change to the state next scene.
- *Menu*. The game menu is rendered in the screen. At the end it goes to different stages depending of the action chosen. If we want to load a saved game it changes to load stage, else it changes to the playing stage. In the case the user chooses to exit from the game, the game ends.
- *Next Scene*. A change of scene can be triggered in <e-Adventure3D> either by the use of an active door (closed rooms) or by the launching of a <trigger-next-scene> effect. If the next scene is a conventional scene then the scene is rendered, the camera changed to the default camera of the new scene, and then goes to Running Effects stage. Otherwise a cut scene is launched so the stage changes to video or slide stage. In all cases the garbage collector is called in order to free system resources.
- *Next chapter*. A change of chapter can be triggered in the same cases of a scene change, but using the effect <end-chapter>. A loading screen is shown and the same things that are done in the stage next scene are done in this stage using the first scene of the next chapter (according to the next chapter in the descriptor document). If there are no more chapters the game ends.
- *Conversation*. When a <trigger-conversation> effect is executed, the stage is changed to Conversation. In this stage the current conversation is shown and its tree/graph analyzed along the user selects different options. Once the conversation is over, control is returned to the running effects stage.

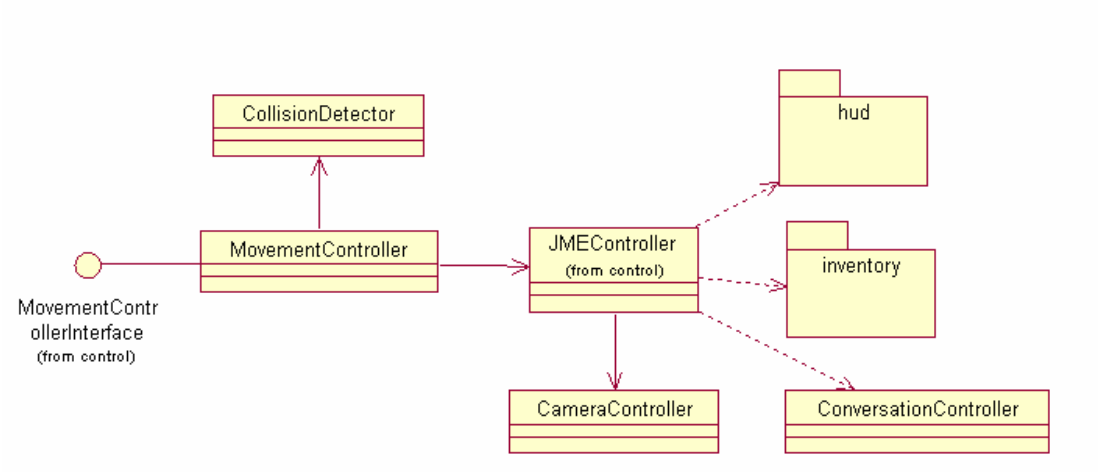
[5] For simplicity we will refer only to the RUNNING_EFFECTS stage, but if the effects were launched from a conversation the flow will return to the RUNNING_EFFECTS_FROM_CONVERSATION stage.

- *GC*. This is the stage used to call the Java garbage collector.

Let's see a state diagram for the stages we showed before, it illustrates the transitions between the different stages:



And last, but not least, *JMEController*. Maybe its name is not enough descriptive, as it not only controls the Graphic User Interface through the JME (what the user sees). It is also entrusted to control all the actions that are indispensable for 3D Games, but that not affect our "state" of the game. That is the control of the movement of the player (including collision detection), camera handling, and of course, is in charge of changing the view of the HUD, conversations, books, slides, videos and inventory panels when it is needed. As we have just told, each of this elements are updated by the JMEController but through a sub controller stored in the same package of JMEController: *core.control.controllers* (there is a sub controllers for the HUD called HudController, another for the inventory called InventoryController, etc)



4. Editor tool

4.1 Rules and guidelines.

As we did with the engine design, some rules and guidelines should be followed in order to do a tidy implementation process and make easy to add new characteristics in the future to the editor tool.

4.1.1. The MVC pattern

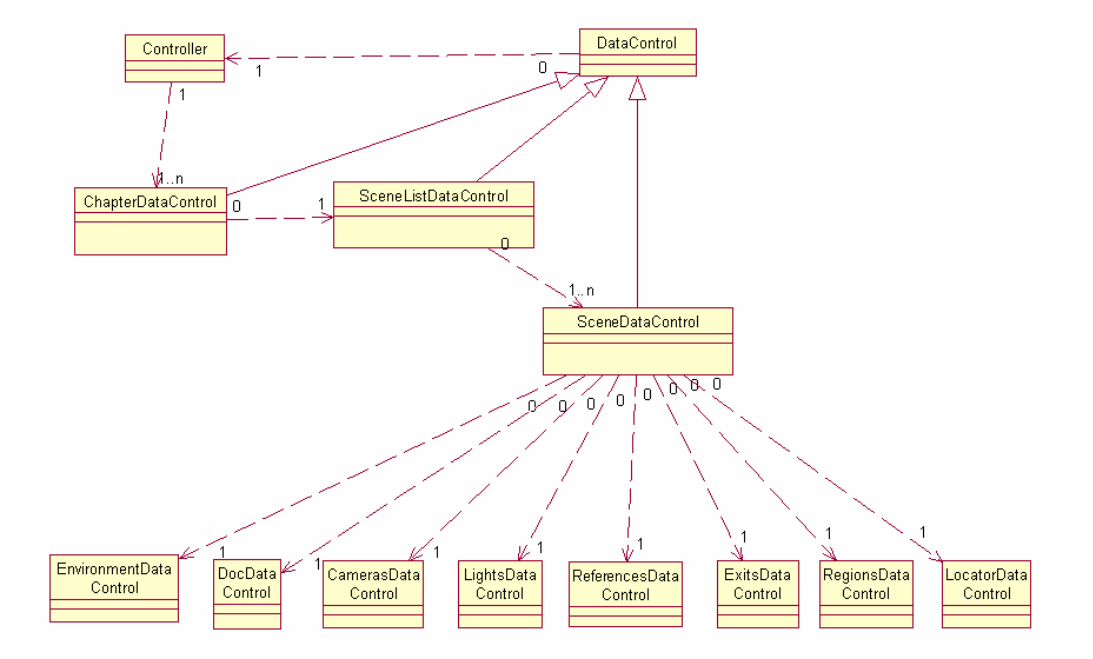
The editor tool design should be also directed by the Model-View-Controller pattern.

Controller

The *Controller*, which is located in the package *editor.control,* is in charge of changing the data and show the view according to the will of the editor tool users.

The package *editor.control.controllers* contains some sub controllers linked to the main controller, these sub controllers are designed for more specific porpoises. They take care of the different parts of an adventure edition: there is a controller for the scenes, another for the items, etc. And there is also another layer of sub controllers linked to the first sub controllers designed for more specific porpoises; for example, the scene controller uses a controller for the cameras, another controller for lights, etc. These controllers hierarchy can seem like something very complicated but it makes sense when you understand the great amount of work each controller has to consider. Moreover, the independence between these controllers and the specific work they are supposed to do make easier the work process when two developers are programming at the same time.

The next class diagram tries to illustrate how the hierarchy of controllers is organized. We have chosen a small piece of the hierarchy where we show the controllers used to manage the scenes. The main controller (class Controller) delegates in one controller for each chapter. Each of these chapters can have one or more scenes and a scene controller is needed for each one. Scenes are very complex so we divided the control of the scenes in some other sub controllers such as the controller for the environment, the controller for the cameras, etc.



There are some extra packages that are used by the Controller: the package *control.assetscontroller* which is in charge of the art assets and the package *control,writer* that hold the classes in charge of writing the XML files from the data generated with the editor tool. This is a remarkable difference between the editor tool and the engine where the data contained in the package *common.data* does not change along the game (it is only read).

As it can be imagined, in order to load adventures the engine uses the objects of the classes contained in the package *common.loader*.

Model

The adventure data is kept in the package *common.gamedata* shared with the engine. The package *editor.data* only contains some metadata used to keep the consistency of the adventures.

View

The view of the editor tool is represented by a window (called MainWindow) which is defined in the package *editor.gui*. The window is divided into three parts: the tree that contains the elements of the adventure which is being edited, the editor menu and the panel with the information of the element selected in the tree. When users select a part of the adventure in the tree, the controller creates the panel associated with this part of the adventure and it is drawn in the corresponding part of the window. It can be seen that there must be one sub controller for each of the panels in the package *editor.gui.elementpanels*. This is because the events produced by users in the panels must be captured and treated by the sub controller linked to this panel. The last diagram showed that there is a

class called DataControl. The class DataControl is an abstract class extended by the controllers of the elements that will be represented in the tree of the main window. So that DataControl has methods to manage the tree and the identifiers control.

4.2 The design package by package

4.2.1 The class EAdventure3DEditor

EAdventure3DEditor is the very first class involved in the editor tool execution. It shows the start dialog so users can choose between loading an edited adventure or create a new one. Once users choose, the Controller is called to initialize all parameters and it takes the control of the edition process.

4.2.2 Package control

The class Controller controls the execution of the editor tool. Let's detail the most important packages inside the package control because these are the packages which are used by the class Controller in order to control the execution:

- *control.controllers*: this package contains all the sub controllers that manage the data and the view of each specific part of adventures.
- *control.assetscontrol*: contains the classes that control the assets included in the adventure.
- *control.writer*: the class Writer of this package is in charge of writing all the XML files that compound an adventure using DOM. It will write documents based in the data that can be hold following the classes of the package *common.gamedata*

4.2.3 Package data

It contains the package *supportdata* which contains a class that stores all the identifiers used in an adventure (IdentifierSummary) and another class with all the information related with the flags (EditorFlagSummary). These two classes help to keep the adventure consistency. Of course, the real data definition is contained in the classes of the package *common.gamedata*

4.2.4 Package gui

The view is very important in the editor tool and this package contains all classes related with it. The main class is the class MainWindow which extends from the java swing JFrame. As we have explained before this window is divided in three main parts: the menu in the top, the three in the left and the current panel in the

right part. The menu is defined inside the class MainWindow, but the tree and the panels are defined outside:

- The tree: it is defined in the class TreePanel of the package *gui.treepanel*. All parts of an adventure (scenes, books, conversations, etc) need a representation as a tree node so they can be referenced. The nodes are stored in the package *gui.treepanel.nodes*, each node need an icon and the data of the adventure part that represents.
- The panels: Panels are stored in the package *gui.elementpanels*. This package is divided in other packages with the panels that represent each part of the adventure. The packages *gui.commonpanels* stores the panels that are used in more than one panel, for example the panel with the transformations tool which is used in the item panel, player panel, etc. The JME preview windows that can be seen in many panels are created using the packages *gui.tdrpanel* and *gui.tdrpreview*

There is another package called *gui.editdialogs* that contains the dialogs that will pop up in the editor tool. They are called editor dialogs because users always will have to fill in some information. Examples of edit dialogs are the dialogs used to define an effects, a dialog to edit some light parameters, etc. There is another type of dialogs to show things to users (for example, if users edit a conversation they can preview it), this dialogs are stored in the package *gui.displaydialogs*

The package *gui.assetschooser* stored some java AssetsChooser modified. We use these AssetChoosers to allow the user to choose files of the type we want whenever we want and disallow the user of choosing the wrong files. We also modify these choosers so users can preview the asset before choosing it. For example the class AudioChooser allows the user to choose only files with the extension OGG or WAV and before choosing one, they can play the sound to check if this is the asset they are looking for.

Chapter VIII

Implementation

In this chapter we write about the points we believe more important about the implementation. At first, we explain the technologies involved to implement the whole platform. Then we write about the engine and the editor tool detailing the parts we think are more complex. At the end, we explain an important task of the platform which is the resources management.

1. Technologies involved

The <e-Adventure3D> engine processes the XML documents and the art assets automatically to produce running 3D games. Then a tight connexion between a XML parser system, a 3D games engine (top-quality graphics are not required as it is not a commercial game, but a black box able to manage for us the main problems inherent to 3D games was needed) and the <e-Adventure3D> core is required. Such premises led us undoubtedly to choose a Java-based environment, just as its predecessor. Thus the programming language is Java version 1.6.0_02 and the SAX library of Java is used to parse the XML document. The 3d engine chosen is JME 1.0, due to its simplicity of integration in Java environments.

The <e-Adventure3D> editor was developed using Java technologies as well as the engine. The code has been written in Java version 1.6.0_02, the same SAX parser is used to load adventures and JME engine is in charge of showing the different parts of the adventure which is being edited by the user. We only used another technology to generate the XML files when the user saves an adventure; in this case we used DOM which is a library of Java.

Now, we are going to explain which options we had had for each function and why we decided to use these technologies and not the other possibilities.

1.1. Programming language: JAVA

The election of Java as the programming language was clear: <eAdventure2d> was developed using Java and there were a lot of code lines we need to reuse in our project (because <eAdventure2d> language is similar to the new one). Moreover, it was the programming language we are more familiar with and it is also powerful to cover all the project necessities. In addition, the main reason to choose the Java environment is to be stated: its platform independence and facility to integrate in web systems. That is definitely a requirement, as <e-Adventure3D> is supposed to be later integrated into larger e-Learning systems.

1.2. XML parser: SAX

Two of the most popular alternatives for the XML parser in Java are SAX and DOM and each of them has their own special features that make them very different. SAX was chosen in <eAdventure2d> and we support the election not only because it was easier to us, but because its properties fix better with the project necessities. DOM keeps all the XML information in memory using a tree data structure. The storyboard of an adventure can be really long so, using DOM, would consume a lot of system resources unnecessarily because the information from the XML file must be changed to generate the adventures so it is useless in a tree data structure. However, SAX allows processing the information while it is been parsed and this is what we really need. So, we chose SAX instead of DOM for this task.

1.3. Representing XML files: DOM

The DOM library of Java (JDOM) allows an easy way of representing XML files and it was exactly what we needed. We did not look for any other alternatives to JDOM because it allows us to generate complete XML files using Java characteristics and the way of programming it cannot be less far from complex.

1.4. The 3D engine: Java Monkey Engine (JME)

We could say that JMonkey was our personal choice, because it is the 3d engine so it was not used in <e-Adventure2D>.

There are some positive features we considered on the choice of this engine. Firstly, JMonkey is totally integrated in Java; it is part of Sun's Java.net software, and the programming style is similar to the rest of the Java language. This feature was essential for the project if we were going to use Java, but there were some others that were also important in order to achieve <eAdventure3d> goals: JMonkey supports model loading in a big range of formats with textures, allows user interfaces using Swing components, takes care of the use of multimedia files, and finally it supports the use of keyboard, mouse and game pads.

However, not all that glitters is gold, there were some problems, because JME is a very young technology. One of them is the poor technical documentation; there are no books, just a few tutorials. Lucky for us the official forum in the JME web page (http://jmonkeyengine.com/) is full of information and people ready to help.

The other great problem is that the JME engine is in development process, so some of the features do not run as they are supposed to do. For example, during the development of <e-Adventure3D> we got some problems with the game pad input system.

On balance then we thought that the positive features are more powerful that the others, so we finally chose JME as the 3d engine of <e-Adventure3D> project.

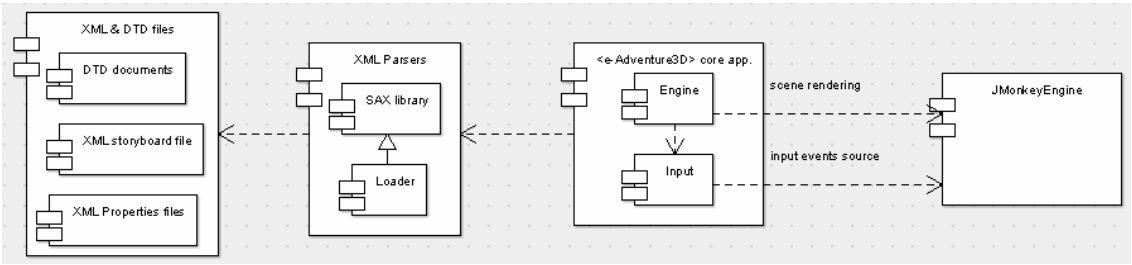
1.5 Additional java libraries

Despite of the libraries added to the project in order to use JME we needed to use some additional libraries to support some additional features. Next we are going to explain these features and the libraries used:

- The EA3D file: this file type is described in a very detailed way in the last section of this chapter, so here we are just going to say that this file type is a renamed ZIP file. Then, in order to manage it in the java code, we added the library called *'truezip-6.jar'*.
- Videos: JME does not support the use of videos into the games, so that we had to use the libraries *'jffmpeg-1.1.0.jat'* for MPEG files and *'jmf.jar'* for AVI files.
- Sounds: JME supports sounds but we needed some libraries to listen to sounds in the editor tool: 'tritonus_jorbis.jar' for OGG files and 'tritonus_share.jar' for MIDI files.

2. The engine implementation

2.1. Component diagram



The component diagram above presented depicts the main structures found in the architecture of the engine. The principal entity is the <e-Adventure3D> core, which contains as well two components of vital importance: the *Engine*, entrusted to convert the input data (whatever the source is) in a functional adventure game, and the *Input,* which due to its complexity has been dedicated its own component

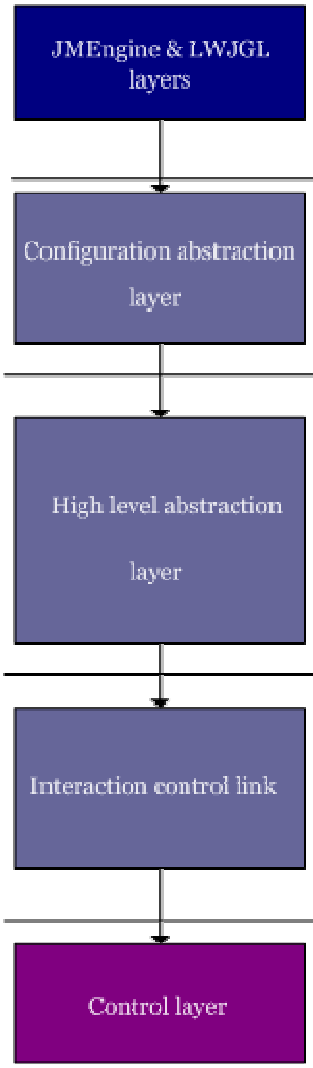
(see section 2.3) and provides the Engine with data coming from game pads and keyboard, being that data already abstracted from the configuration. But Engine does not only gather data from Input; it also uses the *XML Parser* component to get the XML documents (namely the storyboards, adaptation and assessment profiles and properties files) of the application correctly parsed and transformed into Java-based containers of data.

As shown in the figure, the XML Parser has been implemented using the SAX library supplied in the latest Java release. The *Load* component extends that library, providing an interface able to manage (i.e. load and store) <e-Adventure> XML documents.

The last entity deserving a special mention is the JMonkeyEngine. It provides abstraction not only for image rendering but also for the most common issues regarding 3D games (collision detection, models transformation, dealing with multimedia files, etc.). It also supplies *Input* with the input signals coming from both keyboard and game pads.

2.2. Input system implementation

Due to the excessive simplicity of the input system provided by JMEngine, a layered system had been implemented on top of it, transforming the simple input events provided by JME in a solid input management system, easy to use by the control layer of the application, as the next figure depicts. Within the following lines the function of each layer is described.



The first layer is the one provided by JME, which deals with the O.S. and openGL libraries through the LWJGL library.

The output data of this layer is a float value for axes and true (when pressed) or false for buttons.

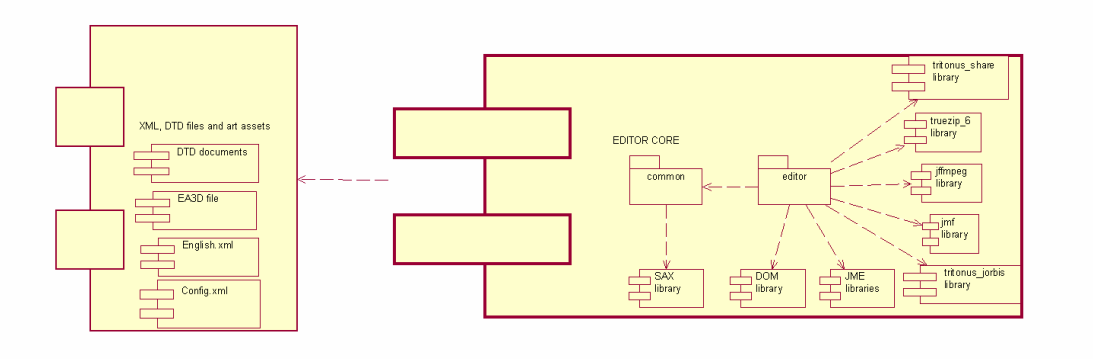
The configuration layer takes the data generated by the JME layer, adding information about how the user's game pad is configured. It means that, thanks to the stored settings, when an input event is produced it is identified which button/axis of the "<e-Adventure3D>" game pad produced it.

Thus far it is possible to retrieve the state of the input in terms of its values. However, a higher abstraction is required, transforming float and Boolean values into **Events**, which are the output of this stage. Therefore it is possible to detect button pulses (i.e. a button has been released or pushed down) or if they still pressed or released. In addition, this layer provides a simple mechanism to identify complex combinations of input (both button and axis) events.

This layer establishes, for each type of interaction (list shifts, movements, etc.) the combinations of input events will trigger a certain action in the game. For instance, it is set that the circle button (only when pressed) triggers conversation interaction. Therefore this layer links the input system with the control layer of the engine, which decides how to update the game when a certain combination of input events is produced.

3. The editor tool implementation

3.1 Component diagram



The component diagram above represents the main structure of the editor tool divided in two different components.

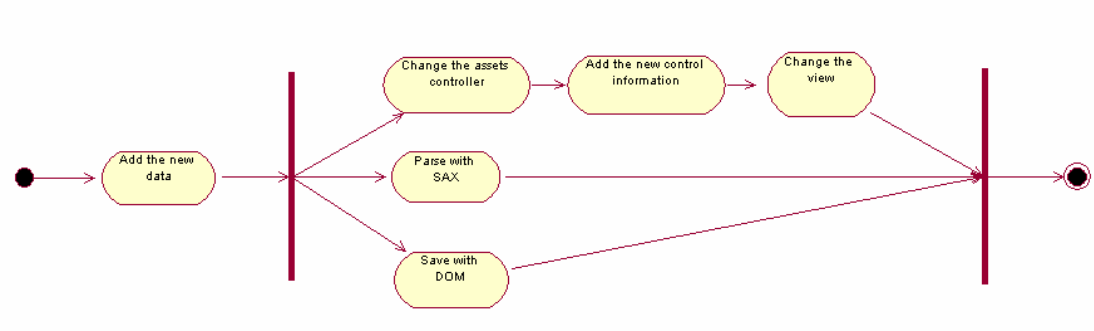
The component in the left contains all the data needed to run the editor tool. It includes the DTD files (adaptation.dtd, assessment.dtd, descriptor3d.dtd and eadventure3d.dtd) used to validate the XML documents stored in the ea3d file, which also contains the art assets. The file called 'English.xml' is an XML property file that contains the information in English language that is shown in the editor (for example in labels, buttons, borders, etc). It is very useful because it can be translated to any language in order to change the language of the editor. 'English.xml' is referenced in the file 'Config.xml' and, for example, if we have translated it to Spanish in a file called 'Spanish.xml' we just have to change the reference in the 'config.xml' file in order to change the editor tool to Spanish. 'Config.xml' is also an XML property file and it also keeps information about the files recently opened and some other characteristics of the editor tool.

The component in the right defines the editor tool programming code used. This code references all files mentioned in the left component. As we have told before, the entire editor code is divided in two packages called common and editor. The editor code also references java code stored in the libraries represented in the diagram (the function of each library is detailed in the section 'technologies involved' of this chapter)

3.2 How to extend the editor tool

This section explains how to add new characteristics to the editor tool. It is useful not only if someone needs to extend the editor, but to understand the implementation (this is why we decided to explain it here).

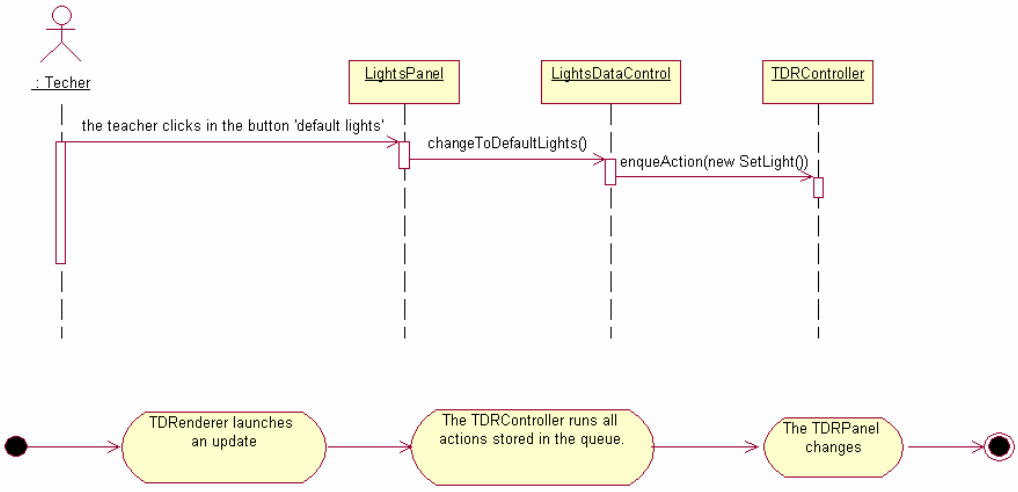
In the next activity diagram we will see the steps in general terms:



Then we are going to explain these steps with an example. Let's imagine <e-Adventure3D> doesn't support the edition of lights and we want to extend the editor tool.

- Add the new data: Firstly, we must extend the DTD in order to support lights in the <e-Adventure3D> language. Secondly, we should add the necessary data in the package *common.gamedata*. We add a new package called lights with the classes needed into the package *common.gamedata.scene.view*, because this is where lights belong according to the DTD that defines the <e-Adventure3D> language. We also need to add the lights data to the scene data, so that we should use an array of lights or something like that.
- Change the asset controller: Sometimes will be necessary to change the class *AssetController* (package *editor.control.assetscontrol*) when a new type of art asset is used for the new characteristic or when an existing art asset is loaded in a different way. The used of a new type of asset (with a new file extension) will also turn into developing a new asset chooser (package *editor.gui.assetchooser*). This is not the case for the lights so this step can be omitted.
- Add the new control code: We must create a new sub controller to take care of the new characteristic. In the case of the lights we should add a new class that we call *LightsDataControl* which must be kept in the package *editor.control.controllers.scene*. As the lights are part of the scenes we must link the new controller to the scene controller which is called *SceneDataControl*. At the same time, the scene controller is linked with the chapters' controller and the chapters' controller is linked with the main controller (represented by the class Controller of the package *editor.control*). This is how the hierarchy of controllers (explained in the design) works.
- Change the view: We must develop the new panels that represent the view

of the new characteristic. These panels will be controlled by the controller we have just created. So we create a new class called *LightsPanel* and add it to the package *editor.gui.elementpanels.scene.locator.view* because it is a part of the scene panel in the view. If the new characteristic is independent from the others (for example, if it is a new kind of scenes such as the cut scenes) then we must create a new tree node to represent it in the tree of the main window (the tree nodes are stored in the *package editor.gui.treepanel.nodes*) and link the new node to the chapter tree node. When the new characteristic needs to new use of the JME-based preview window (class *TDRPanel*) we must create a class that extends the class *TDRAction* and store it in the package *gui.tdrpanel.tdractions*. These actions change the preview, there are actions such as change the camera add a model to the scene, select an element, etc. The actions executed by a controller called *TDRController,* which will make the appropriate changes on the *TDRPanel*. This controller owns a queue of actions that are applied to the preview window following the order of the queue every 'update' (the time between updates is controlled by the JME). In the lights case we must create a new action called *SetLight* which its function is to change the light in the scene which is being preview at the JME-based preview window. Let's see a graphical example:



Firstly, we can see a sequence diagram that represents the consequences of a user selection. If the user (the teacher) clicks on the 'default lights' button of the panel LightsPanel (in a scene edition), then the lights controller of the current scene (LightsDataControl) will add the action SetLight to the actions queue of the TDRController. Secondly, the next diagram shows the automatic operations made in order to update the

JME-based preview window. The TDRenderer has an update method (simpleUpdate()) called automatically by the JME where it calls the method 'runAllActions()' of the class TDRController. So that, the TDRController launches all the actions stored in the queue (this actions change the TDRPanel data). Then the TDRenderer called automatically the method 'doRender()' in order to repaint the TDRPanel (according to the new data). We divided it into two different diagrams because the sequence diagram can not show the automatic process drawn in the second diagram. This is because the calls to the methods simpleUpdate() and doRender() of the class TDRenderer are called automatically by the JME every cycle.

- Parse with sax: the package *common.loader* contains the classes used to parse the data from the XML documents and store it in the data classes (this is used to load an adventure). So that, we must extend the class *SceneSubParser* (package common.loader.subparsers) because lights are a part of the scenes in the XML document and they must be also parsed.
- Save with DOM: Some classes of the package *editor.control.writer* will need to be extended in order to generate the XML documents (in order to save the adventure) from the data classes (package *common.gamedata*). In the case of lights, we must extend the class *SceneDomWriter* of the package *editor.control.writer.domwriters* because lights are a part of the scene according to the DTD.

4. Design and implementation details of the resource management: the <e-Adventure3D> file

4.1. The troublesome of distributing the games

So far we have widely discussed about the xml file which rules any <e-Adventure3D> game. As it has been clarified, every single element defined in the game (i.e. player, characters, items, scenes ...) is declared on the storyboard file, which is created by hand or automatically with the editor. But this file on its own does not produce a game. A game is composed by this *storyboard file* (or a set of them if more the game contains more than one chapter), a *descriptor* (an xml file containing generic information about the adventure: the title, a description of the game and the set of chapters it contains), and a lot of *resources used to render the game*. Those resources are 3D models, textures, images, sounds... without them

the game would not exist at all.

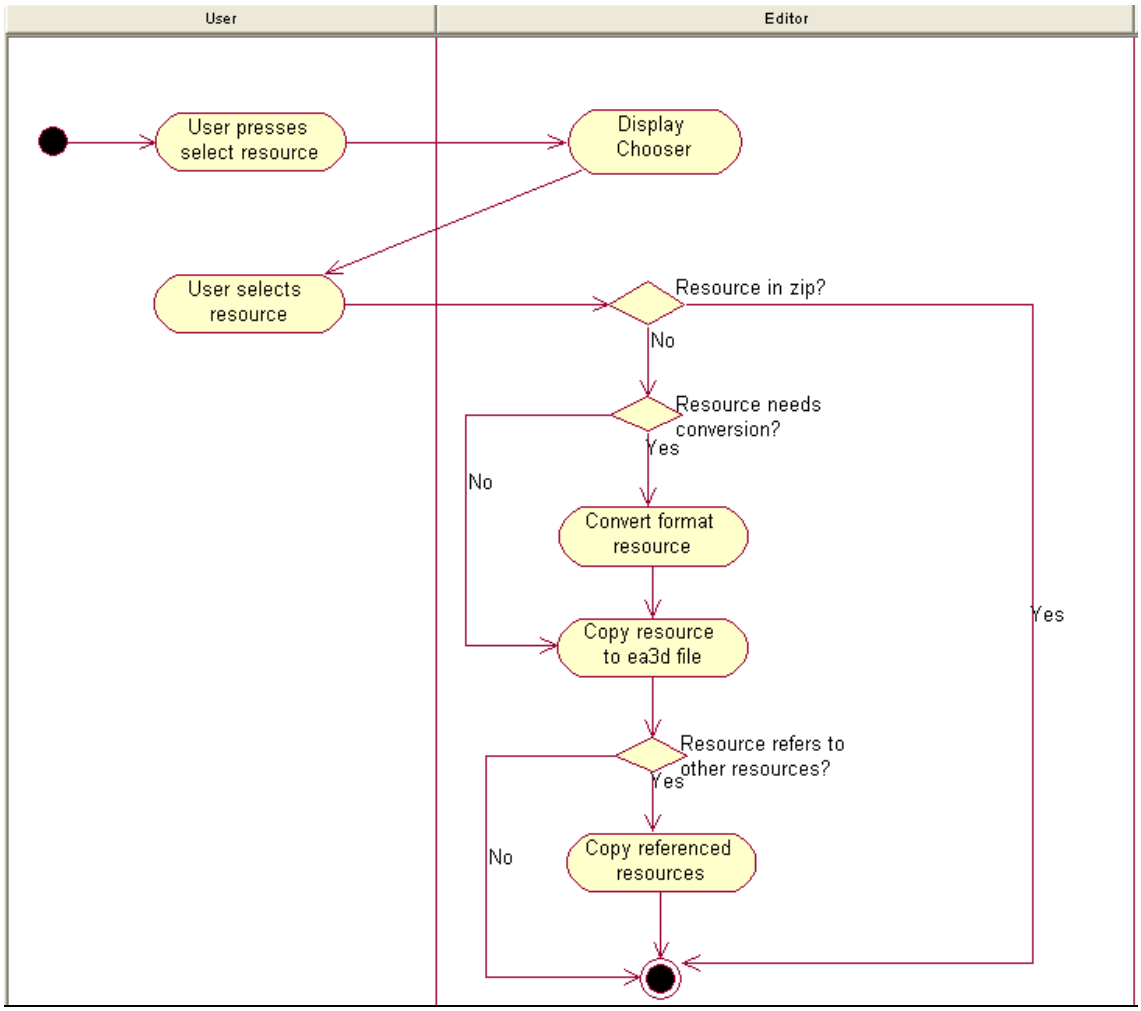
Such complexity of different files makes the development and distribution process of the game hard for instructors. You do not only need to produce a storyboard, but also somehow distribute all the referenced resources along with it. And be aware of absolute paths! In short, a real headache which lowers the real potential of the application, as it is hardly impossible to imagine the distribution of a whole 3D game which resources are separated in different files. Moreover, this situation contradicts the requirement that <e-Adventure3D> games must be deployable from a LMS.

4.2. Packing the contents in a single file.

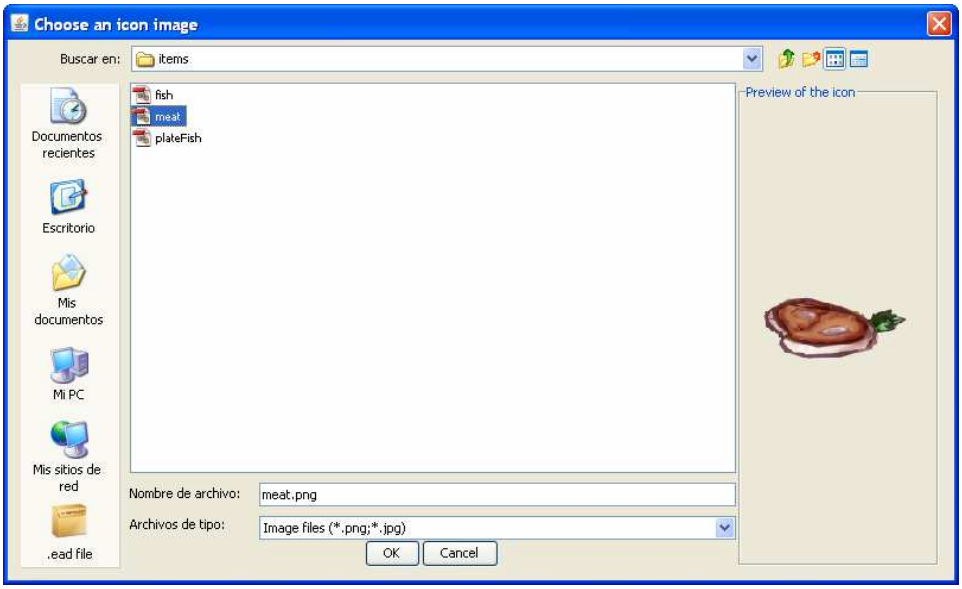
To tackle this situation, all the files referenced in a game are put together into an **"ea3d" file**. That is just a simple zip file which is renamed with the ea3d extension, abstraction what the file really is. Therefore, when authors create a game they only need to distribute a single file which contains all the engine needs to make it run. And, at the same time, solves the problem of the absolute paths: you can specify relative paths to the "ea3d" file, so it does not really matter where the file is.

This process is completely transparent to the user as the editor does it for you. When you select a resource (asset) for the game, it automatically includes it in the ea3d file. Then, when it is ready, the engine loads them from the compressed file with no problems. In this section you will find more detailed documentation about how this is carried out from the beginning (when a new file is created) to the end (when the game is executed) along with the types of resources supported by the editor.

The first step takes place when the user runs the editor and selects to create a new file. Then, the editor prepares a new zip file with the .ea3d extension, copying there some default resources for textures and models, to be used if you do not want to customize them. That is the case of the sky for open environment scenes and textures for terrain, walls and ceiling (for scenes as well). Then, when the user selects a resource some operations are carried out. Here we will describe the whole process, depicted by a sequence diagram.



1. Ask the user. The first task is to prompt the user to choose to which resource wants to make a reference in the game. According to the type of resource (model, audio track, etc.) a different chooser dialog is shown. For instance, the figure below shows the chooser for icon images:



As you see, when a valid resource is selected (in this case PNG or JPG files)[6] , it is previewed on the resource chooser. Moreover, there is a shortcut on the left panel (bottom) to directly access those resources already included in the game. This facility is useful for the rapid reuse of those resources which are referenced frequently. In addition, performance is enhanced as some resources need to be converted before they are included in the game file, as you will see within the next paragraphs.

2. Validation. Once the resource has been selected and validated, the application checks if it is already in the game file (ea3d). If this check is affirmative no further operations are needed. Otherwise, the application gets the resource ready to be copied to the game file.

2.1. Format conversion. For some kinds of resources (e.g. models) a format conversion is required. There are two reasons to force that conversion: improve the performance of both editor and engine, and on the other hand get the range of supported formats broadened. Let's give an example of both cases.

Models

Models are usually large files distributed in a wide range of formats: 3DS, MAX, MD2, OBJ... and the list grows and grows. Although our 3D engine (JME) supports some of them, it has its own binary format, which is what really understands. That is, when a non-jme model is loaded, it needs to be converted first, which is a high-cost operation that lowers the loading times of the engine. To solve that, when a

[6] See section 3.3 of this chapter for more details about what assets are supported for any kind of resources

model is added to the game file it is firstly converted to the jme binary format, so it only needs to be converted once.

Audio

Unfortunately, JME only supports WAV and OGG audio files. However, the most widespread audio format is obviously MP3. For that reason, our editor is entrusted to automatically convert the audio track to the MP3 format.

3. Copy. Once the resource has been properly converted it is copied to the game file. During this process, the application monitors if the resource makes references to other secondary files. That is the case of model files, which usually define skins on separate texture files that need to be included. Then, those files are copied along with the main resource file so they are finally ready to use.

4.3. Table of resource types and supported file formats

In this section it is described the kind of resources you can use in an <e-Adventure3D> game, and the supported file formats for each one.

Models

A model is the 3D representation of any element in <e-Adventure3D>, as it is in conventional 3D games. The next formats are supported by the editor, which converts them to the JME binary format.

File extension	Format
3DS	Autodesk 3D Studio format. NOTE: MAX files are not supported
OBJ	Open file format developed by Wavefront Technologies. Most of professional tools supports this format: Autodesk Maya, Poser ...
MD2	Quake II model files format
MD3	Quake III model files format.
MS3D	Milk Shape 3D binary format

Restrictions

1) Although all of these are supported, we encourage you to use 3DS and MS3D files, as other formats have not been enough tested to ensure they work properly.

2) Models are allowed to refer to some secondary resources, as skins, animation files, etc. However, the model must be contained in a single file. Composed models, as conventional Quake III models[7] are not supported.

Images

Images are used in <e-Adventure3D> in the next contexts:

Icons. An icon is the image that will be displayed on the inventory when an item has been grabbed.

Textures. Textures are used in <e-Adventure3D> to customize the appearance of scenes and models.

Slide-scenes. <e-Adventure3D> supports inclusion of slides presentations, which basically are successions of images.

In all cases the supported formats are:

File extension	Format
PNG	Portable Network Graphics format
JPG	NOTE: All JPG images must have the JPG extension.

Video-scenes. <e-Adventure3D> supports the inclusion of videos as well, which can be in one of the next formats:

File extension	Format
MOV	QuickTime® video format

[7] The Quake III Engine assembles the model of players taking three separated files: head.md3, upper.md3, lower.md3

AVI	Multiple formats are stored in AVI or MPG extension. To check which formats and codecs are supported, consult the online source: http://java.sun.com/javase/technologies/desktop/media/jmf/
MPG	

4.4. Structure of the ea3d file

Finally, we will describe what is inside an ea3d file. This will be useful for those users who desire to produce games on their own with no help of the editor, and for debugging purposes.

As you already know, an ea3d file is in fact a zip file. Therefore you can use a zip tool such as WinZip, WinRar, etc. to access its contents. You can uncompress it, modify its contents and re-zip it again (renaming it to .ea3d) and it will work. But be careful, modifying this file by hand could damage.

If you open an ea3d file this is what you will see:

A descriptor file (*descriptor.xml*). This syntax is strict, so you cannot change the name of this descriptor or neither the engine or the editor will be able to read it. It contains general information about the adventure and links to the xml files where every chapter is defined. Please ensure you put all the xml files of the chapters in the ea3d and that they are properly referenced in the descriptor file. Otherwise the game will be unplayable and maybe uneditable.

The DTD for this document is descriptor3d.dtd.

The chapter files. You can name these files as you desire, but ensure they are correctly linked in the descriptor file.

The DTD for these documents is eadventure3d.dtd

Resource folders. The editor puts all the resources of the same type in a folder. You can organize them as you wish, as long as the adequate reference to the relative path of the resource is done. For instance, if you are defining an item and you want to put its icon within the ea3d file in a file named “icon.jpg”, in the xml file you will type its uri as “icon.jpg”. If on the contrary you would like to put it on a folder, called “myicons”, the reference would need to be “myicons/icon.jpg”.

The only restriction here is for models. If you go inside the folder assets/model, you will see every model is put in a separate folder. In each model folder there is a file, always named the same (model.jme) which is the file of the model. Then you find all the other files the model requires, as textures or animation files. You cannot change this: every model must be in a separate folder. The name of the resource is the name of that folder, so you will refer to it using that name. Put all the files the model needs in that folder, otherwise it will not work. Then, the references in the xml file will be to the folder, not to the model.jme file.

Chapter IX

Characteristics of <e-Adventure3D>

1. The <e-Adventure3d> engine

The <eAdventure-3D> engine is the unit which processes the EA3D file, which is a renamed ZIP file. Those files contain all the data needed to interpret and execute the game. Therefore the EA3D contains the next entities:

- The art assets: 3D models, images, sounds and videos involved in the game.
- XML documents. These documents rule the entire game, providing indications to the engine about how the art assets must be provided, and about the narrative elements of the adventure. There are four types of XML documents, which language syntax (DTD) can be consulted in the Appendices chapter of this document. Those files are:

  - **Storyboard files.** The narrative flow of the games is structured in **chapters**, small pieces which are independent one of each other, easy to produce, maintain and keep in memory. The narration of each chapter is ruled by a different storyboard file.

  - **The descriptor of the adventure**, which contains general information about the adventure such as the title, description, gui settings and configuration data for each chapter

  - **Assessment profiles**. Each chapter can be attached with an assessment profile to produce an automatic evaluation of the student. The evaluation is defined as assessment rules, which are stored in these assessment profiles.

  - **Adaptation profiles**. Each chapter can be attached with an adaptation profile as well. Adaptation is a feature of the games which allows to gauge its behaviour according to the profile of the student. Analogously to assessment, adaptation is defined in terms of rules, which are stored in these files.

Using these components the engine produces running games.

This part of our project has been developed using Java technologies (technologies are detailed in the chapter called “Design and implementation”).

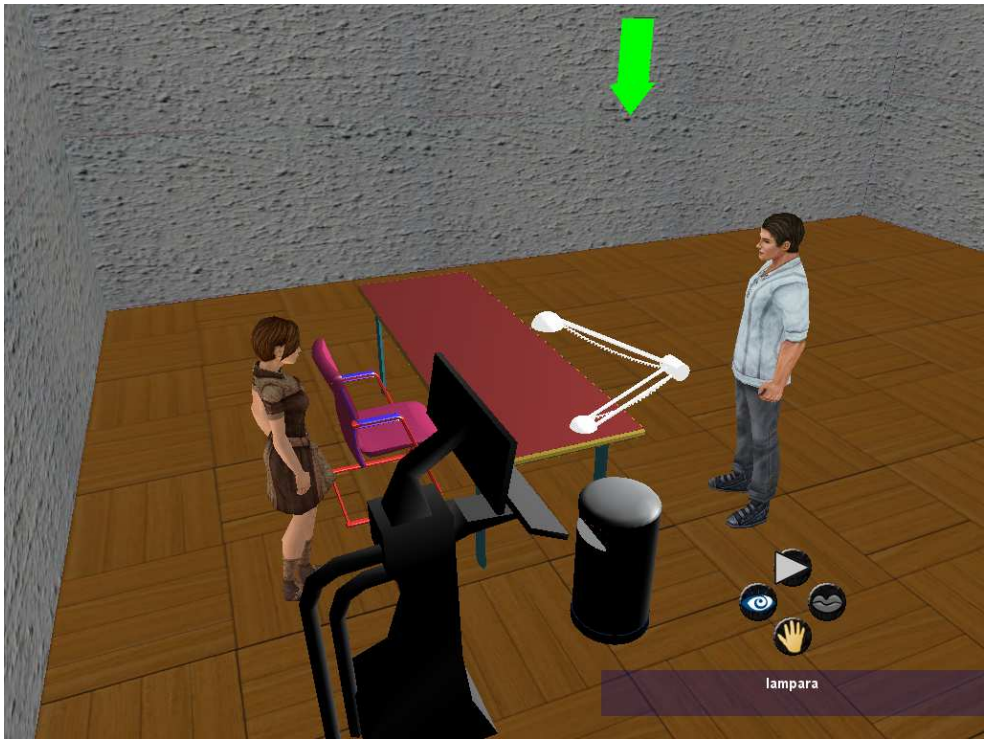
But, how can we get a running game from XML documents and art assets? There

are some features that 3D adventure games must support such as the movement of the player, the information about the elements near to the player (which is displayed by the HUD) , the inventory where the player keeps the grabbed items and the conversations between characters. Once we have the information of the adventure parsed from the XML files, we show it in the suitable moment using JME and allow or disallow the actions the user can perform according to this information.

Let's see the most important parts of our engine:

The **input system** gathers the data about the state of the input devices (keyboard and game pads) and route them to the appropriate controller (i.e. movement and interaction controllers). However, the problem was that each game pad is different so we developed a game pad configuration utility to standardize the correct identification of the buttons. JME is a modern 3d engine of recent creation so it does not have all the functions we need, so we had to dedicate a great effort to the improvement of this system.

One important thing was the design of the **HUD**. It shows information about the items accessible to the player and the actions that can be made to the one which is chosen. There are four possible actions: use, speak, grab or examine. So we thought that these actions could be associated with the four buttons a game pad usually has. We are going to show an example where the player is near to a lamp, the HUD shows the name of the element and gives a clue about the actions that can be made to the item. In this particular case we can grab or examine the lamp so these two HUD icons are coloured. We also paint an arrow over the selected item in the current moment:



The **inventory** shows the items that the player owns at the current moment of the game. The user can see an icon that represents the item and its name. One item of the inventory could be used with other elements (from the inside or the outside of the inventory), if this ever happens the HUD will show it by highlighting the action icon. The item in the middle is the one that is selected:

The **game menu** can be used to change some parameters of the game (such as allow or disallow sounds in conversations or see the subtitles). It is also used to save and load a game or exit of the game.

Between the load of chapters of the adventure there are loading screens to let users now that the engine is not blocked. This is its appearance:

2. The <e-Adventure3d> editor tool

The main goal of <e-Adventure3D> editor is to make easier the way of writing storyboards to be played by <e-Adventure3D> engine. The editor becomes a necessity since the moment when users try to create a game that involves a few scenes.

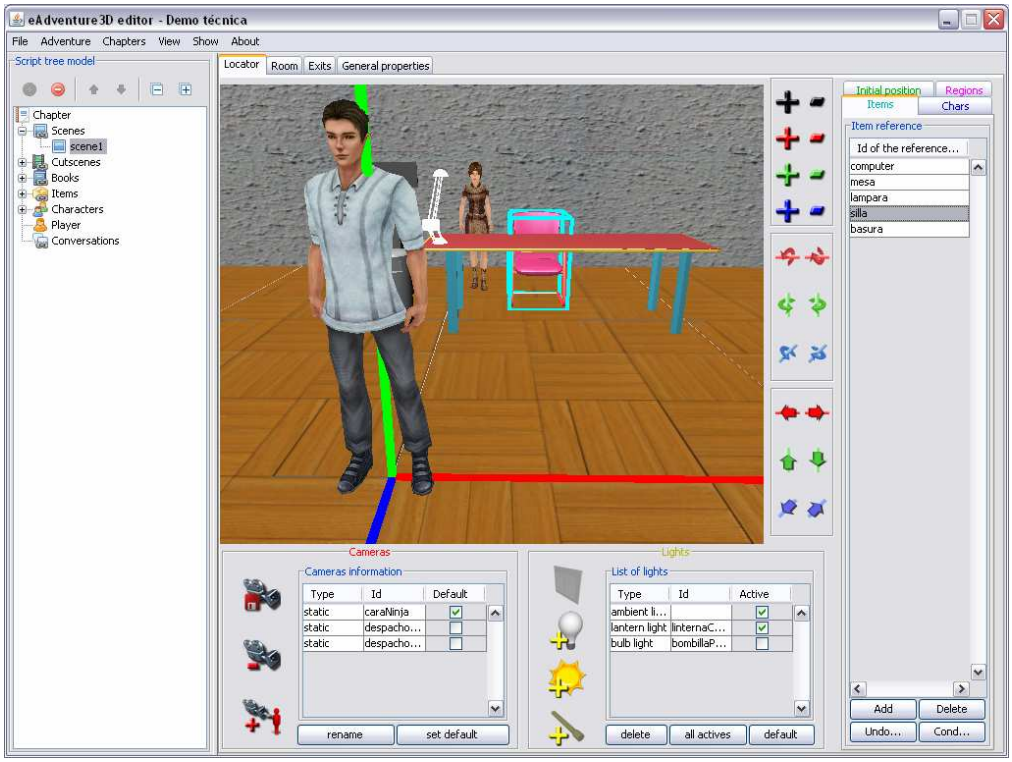
As the engine, the editor tool was developed using Java technologies.

There is a complete compatibility between the editor and the <e-Adventure3D> language; as a result everything that can be written in the XML documents is possible to be done graphically in the editor. Then, if you can have a game using either ways, why have we developed an editor? This question turned obvious when we tried to create our first real test game. Although it had only one chapter with three scenes, twenty items and seven characters each of the characters had a few conversations and both, characters and items, had transformations[8] to be carried out on them. The fact is that the XML file had more that one thousand and five hundred written lines and involved more than a week of work. The <e-Adventure3D> editor solved this problem. It allows game makers to create the adventures in a graphical way so they do not have to know anything about XML or <e-Adventure3D> language and it reduces the amount of work hours remarkably.

As a result the <e-Adventure3D> editor is a great advantage against writing the XML document, even if the user dominates the language. There are some strong points that turn the editor in a very powerful tool:

- JME-based preview windows: The editor has JME preview windows that allow users to preview adventures like they will be seen when users executed the adventure in the engine. They are very useful in many ways. For example, a complete scene of the adventure can be preview and users can choose where to place the instances of items and characters or make transformations on their models to fix them in the scene. In the next picture we can see a scene of the Tech demo. The chair is selected so we can apply transformations on it (the user manual shows detailed information about this process).

[8] *Transformations* are the operations required to locate and orientate an element in the scene. That might include scales, rotations and/or translations.



Users can also move the camera in the JME preview windows with drag-and-drops of the mouse wherever they want.

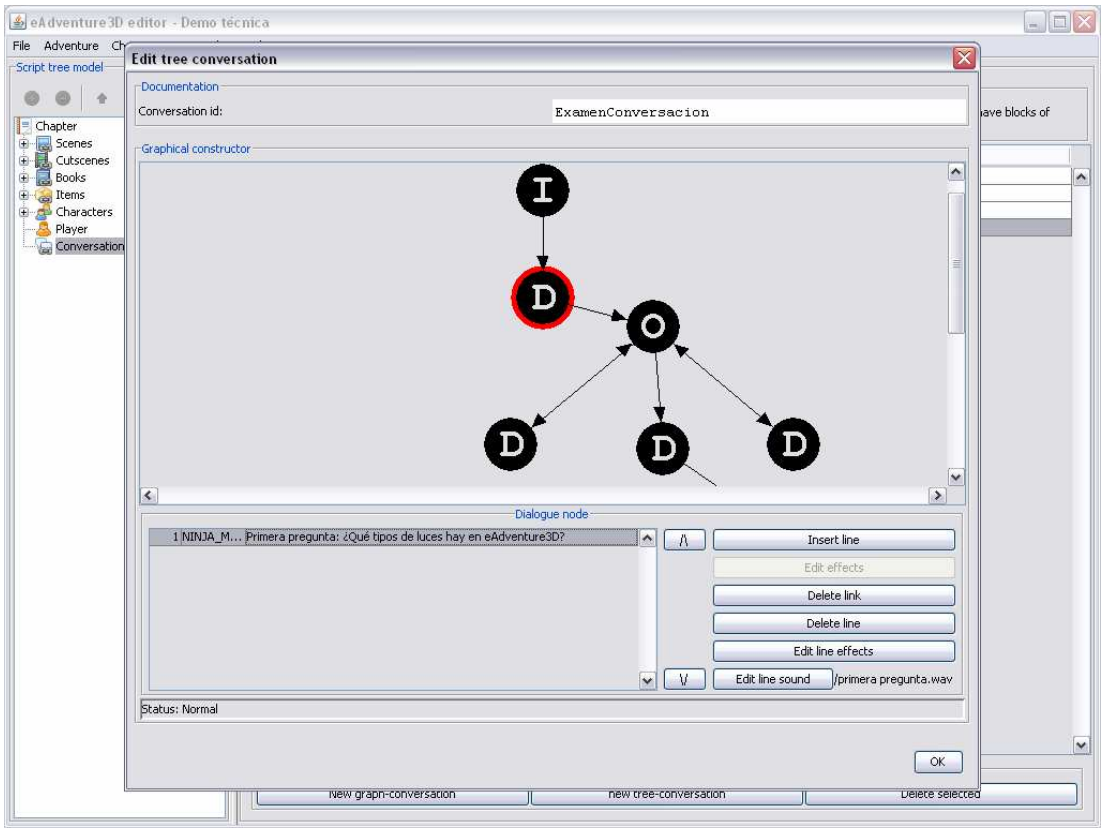
- Cameras management: Users must have basic geometry knowledge to define cameras directly on the documents as it is explained in the next section. This is not a requirement when users create games with the editor, since they can create both types of cameras without any difficulty. As we have written before, users can preview a complete scene and with a simple mouse movement they can place the camera wherever they want and save its position to use it in the adventure with just press the button save camera:

Third person cameras can be created in a similar graphical way.

- Identifiers control: Another difficulty of writing an adventure was to remember all the ids and where we have used them. Ids are used to identify many things such us conversations, characters, items, scenes, cameras, lights and flags and they cannot be repeated. Users must remember the name of items and characters or look them up trough

the document when they want to instantiate them in a scene. If they change the id of a flag[9] they will have to change it every time they have used it in the storyboard. This kind of things requires time and lead to make mistakes. However, not in the editor, where these things are treated automatically: it does not allow users to repeat ids, it deletes all references to an element if it is deleted and allows users to rename an element so all its references will be also renamed.

- Conversations: They are represented in a graphical way with trees or graphs (depending on the type of the conversation). The next image shows the graphical conversations editor that allows users to write in a tidy way a conversation between characters:

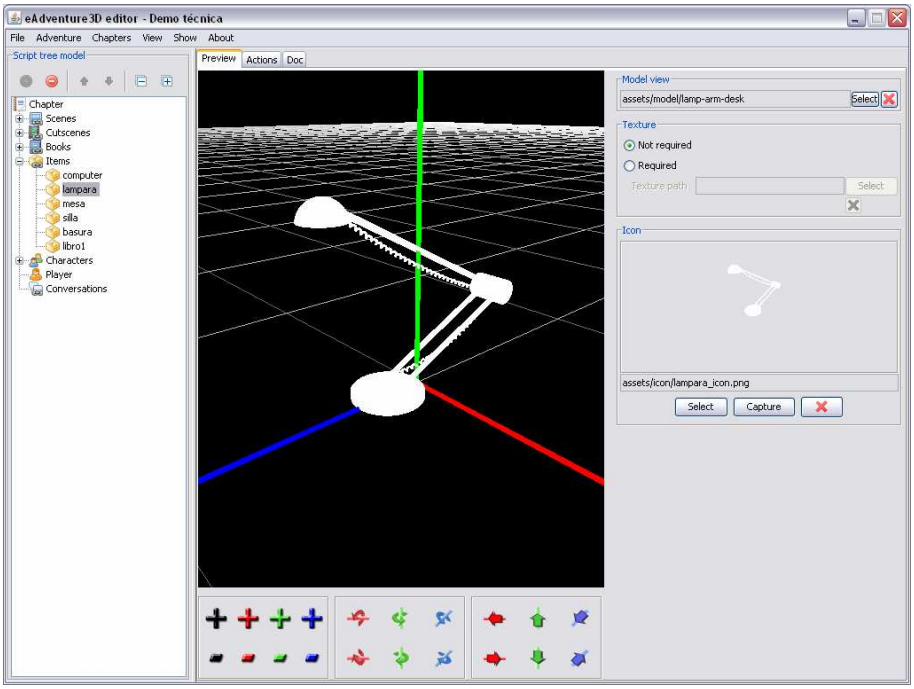


To see a complete description of the conversations, and how to create them (maybe the most difficult point in <e-Adventure3D>) please visit the Appendix A (User Manual). For now just explain that the representation in nodes is the clue which allows the conversation to take different paths depending on some choices of the player. Therefore the nodes are mainly distributed in *Dialog nodes* (which contain the dialog

[9] *Flags* are indicators that can be evaluated as "active" or "inactive" in a certain moment. Those rule the narrative flow of the adventures, as it will be explained later

lines) and *Option nodes* (decision nodes containing option lines).

- Automatic icon creation: There are items that can be hold in the inventory by the player. These items must have a two dimensions icon which represents them. The editor allows the automatic capture of a picture of the item model from a JME preview window and use it as its icon with just a mouse click in the 'capture' button:



Users will not have to produce image assets of each item that can be hold in the inventory. This characteristic reduces the costs of developing a game.

- Assets and XML files are saved automatically: When we add a reference to a new asset while we are creating an adventure with the editor tool, this asset is copied automatically in the EA3D file. Of course this is very convenient because game makers will not have to remember to add the assets referenced into the EA3D file. Moreover, chapters' management can be done with the editor and while we add a new chapter, the xml file is added to the EA3D file too and the descriptor is also updated.

As a conclusion, we think that our project would not be complete without the editor, because, as we have seen in this section, the editor makes the difference: it allows users to define complete solid adventures in less time than writing the storyboard and complicated educational games can be created by users that are not

familiarized with <eAdventure-3D> language, XML or basic geometry.

3. Main features of <e-Adventure3D>

DTD files define the structure of an XML document, so we defined our own DTD for the <e-Adventure3D> language. The idea was to keep the simplicity of the previous project language (<e-Adventure2D>) as much as it was possible. However, in the case of a three dimensional space the only difference is not only that now we need a new coordinate (z) to represent a point in the space. Of course, things are not that simple. There are characteristics in <e-Adventure3D> that turn this task into a very hard one (for example the use of cameras). In our opinion we have simplified the language as much as it is possible without leaving aside the expressive potential of a 3D environment. Moreover, we have developed the editor tool to turn the <e-Adventure3D> language into something transparent to users. In addition, tt was also important that the language would be independent of the JME, so that in the future the engine could be replaced for another one without a big effort.

In this section we are going to review the most important characteristics of <e-Adventure3D>: we are going to explain the language through the DTD file and show some examples of XML parts and its representation in the engine and in the editor tool.

3.1 Scenes

Firstly, we are going to explain the **scenes**. This is the DTD part that defines them:

```
<!--SCENE-->
<!ELEMENT scene (documentation?, resources?,name, environment?, default-initial-position, regions?, exits?, objects?, characters?, view?)>
```

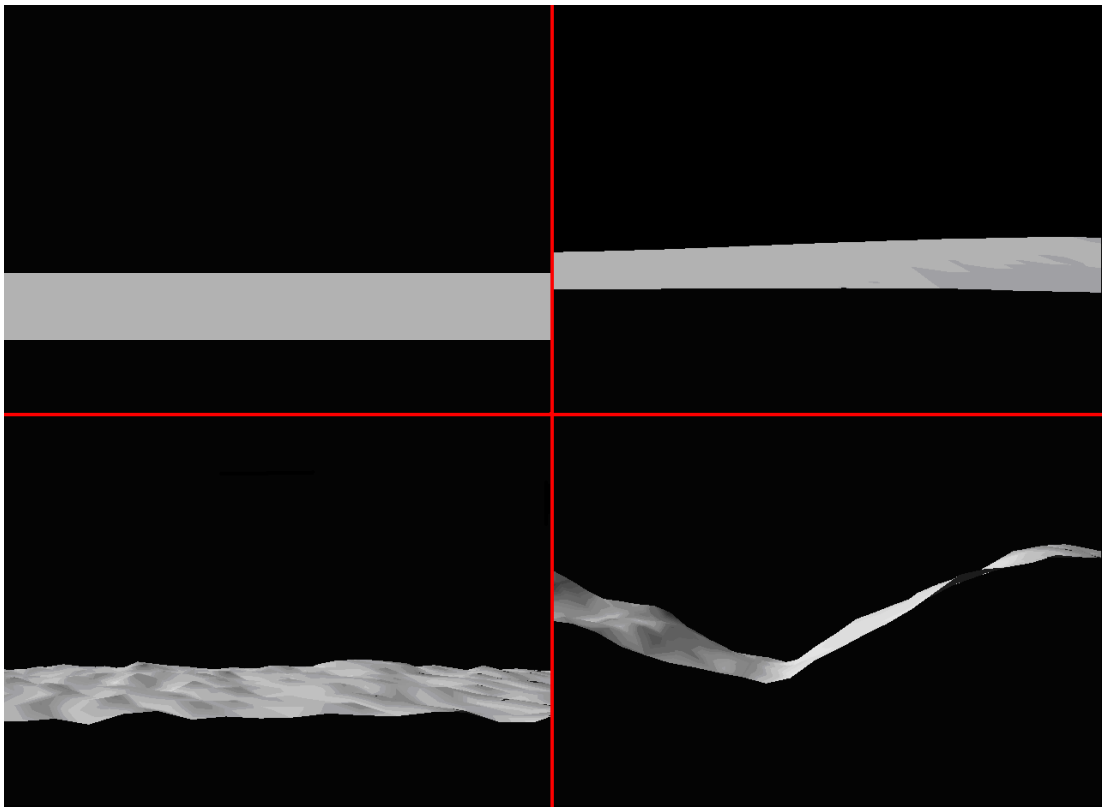
We are going to explain the different parts of the scenes.

One of the simplest tasks of the <e-Adventure2D> was to create the environment of the scene because it was just a background image. A three dimensional space turns it to something much more complicated. Our goal was to simplify this task to users, so (as the DTD depicts) we support the use of two predefined **environments**: *open environments* and *rooms*.

```
<!--SCENE ENVIRONMENT-->
<!ELEMENT environment (openEnvironment | room)>
```

The **open environment** defines a terrain surrounded by a sky.

The terrain has a size, a texture and one of these appearances depending on the roughness selected by the user; it can be plain, smooth, rough or mountainous (they are listed from the less rough to the most rough).



There is also another type of open environment that generates a plain environment with a water texture. The results are quite satisfactory as this kind of environment really looks like a waving piece of sea, and the user does not need to do any extra work to get it. Unfortunately, it is still in an experimental stage so it is not guaranteed to work properly under all circumstances.

On the other hand, the sky it is defined with 6 images so they form a cube that covers the three dimensional space.

The language is defined like that:

```
<!ELEMENT openEnvironment (sky, terrain)>
<!ATTLIST openEnvironment
   fog-enabled (yes | no) "no"
>
<!ELEMENT terrain (terrainTexture?)>
<!ATTLIST terrain
aspect (plain|smooth|rough|mountainous | water) "plain"
width NMTOKEN #REQUIRED
depth NMTOKEN #REQUIRED
uri CDATA #IMPLIED
>
<!ELEMENT terrainTexture (textureScale?)>
<!ATTLIST terrainTexture
%uri;
>
<!ELEMENT sky (north, south, west, east, top, bottom)>
<!ELEMENT north EMPTY>
<!ATTLIST north
   %uri;
>
<!ELEMENT south EMPTY>
<!ATTLIST south
   %uri;
>
<!ELEMENT west EMPTY>
<!ATTLIST west
   %uri;
>
<!ELEMENT east EMPTY>
<!ATTLIST east
   %uri;
>
<!ELEMENT top EMPTY>
<!ATTLIST top
   %uri;
>
<!ELEMENT bottom EMPTY>
<!ATTLIST bottom
   %uri;
>
```

We are going to see an example of an open environment. This is a rough one with a snow texture. Firstly, we are going to write the XML code and then see how it is shown:

```
<environment>
   <openEnvironment fog-enabled="yes">
      <sky>
         <north uri="assets/sky/alpes_north.jpg"/>
         <south uri="assets/sky/alpes_south.jpg"/>
         <west uri="assets/sky/alpes_west.jpg"/>
         <east uri="assets/sky/alpes_east.jpg"/>
         <top uri="assets/sky/alpes_up.jpg"/>
         <bottom uri="assets/sky/alpes_down.jpg"/>
      </sky>
      <terrain aspect="rough" depth="300.0" uri="assets/terrain/scene0.jme" width="300.0">
         <terrainTexture uri="assets/texture/b19nature_landscapes391.jpg">
            <textureScale x="1.0" y="1.0" z="1.0"/>
         </terrainTexture>
      </terrain>
   </openEnvironment>
</environment>
```

Easily we can define another open environment completely different. For example, if we choose a plain one with a sky that forms the skyline of a modern city and we use a grass texture so it looks like a park in the middle of all the skyscrapers, this is the result in the engine:

The other type of predefined environments is the **room**. There are two room types: closed rooms and model rooms.

```
<!ELEMENT room ((standardRoom | modelRoom), sky?)>
```

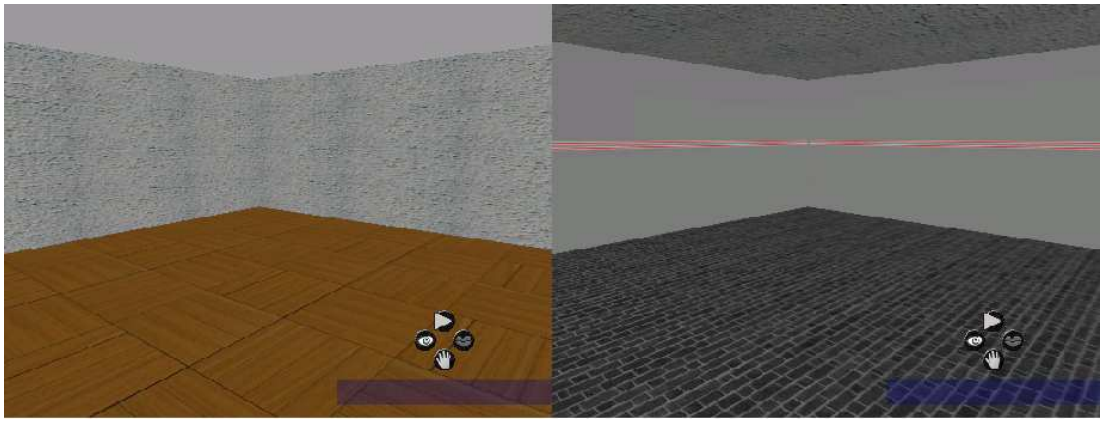
The **closed room** allows users to generate a simple room with the size they specify. It is compounded of four parallel walls, a ceiling and a floor with their respective textures. Users can redefined the default textures set by the editor by just selecting a different image from the file system. In this type of scene users can define from zero to four exits that will be represent in the scene as doors, one in each wall. Rooms are very useful in adventures due to their low production cost, and they change their appearance with only changing the textures.

A room definition must follow the next pattern:

```
<!ELEMENT standardRoom (roomTextures)>
<!ATTLIST standardRoom
   width NMTOKEN #REQUIRED
   height NMTOKEN #REQUIRED
   depth NMTOKEN #REQUIRED
>
<!ELEMENT roomTextures (wallsTexture?, ceillingTexture?, floorTexture?)>
<!ELEMENT wallsTexture (textureScale?)>
<!ELEMENT ceillingTexture (textureScale?)>
<!ELEMENT floorTexture (textureScale?)>
<!ELEMENT textureScale EMPTY>
<!ATTLIST textureScale
%vector3;
>
<!ATTLIST wallsTexture
   %uri;
>
<!ATTLIST ceillingTexture
   %uri;
>
<!ATTLIST floorTexture
   %uri;
>
```

Next we are going to explain a room example. Firstly it is shown the XML. Secondly we can see the same room viewed from the same place with two different textures (change the texture is just change the URI that references the texture).

```
<environment>
   <room>
      <standardRoom depth="300.0" height="100.0" width="300.0">
         <roomTextures>
            <wallsTexture uri="assets/texture/wall.JPG">
               <textureScale x="1.0" y="1.0" z="1.0"/>
            </wallsTexture>
            <ceillingTexture uri="assets/texture/ceilling.jpg">
               <textureScale x="4.0" y="4.0" z="4.0"/>
            </ceillingTexture>
            <floorTexture uri="assets/texture/floor.jpg">
               <textureScale x="4.0" y="4.0" z="4.0"/>
            </floorTexture>
         </roomTextures>
      </standardRoom>
   </room>
</environment>
```



Model rooms are generated from a 3D model. We choose the model that will

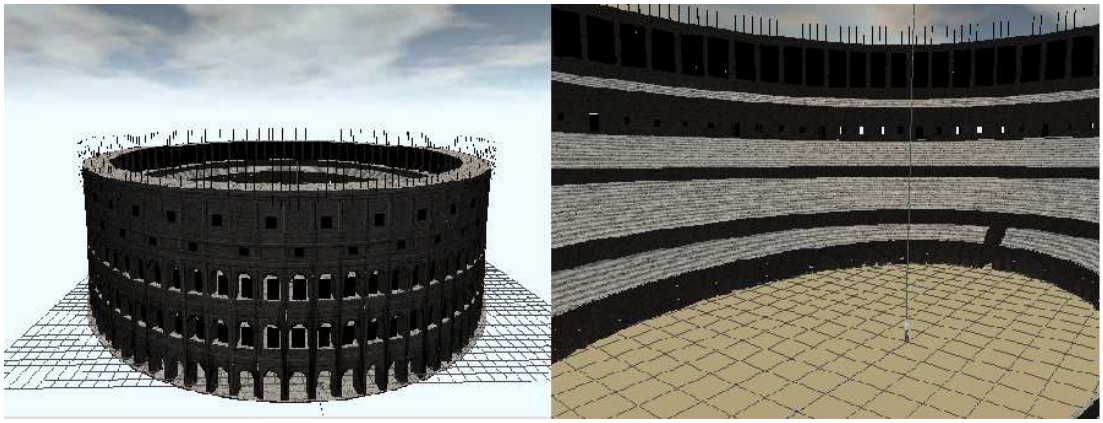
form the room and the region where we want the player to stay. Next we show the DTD definition:

```
<!ELEMENT modelRoom (extension?,transformations?)>
<!ATTLIST modelRoom
   %uri;
   %positionR;
>

<!ELEMENT extension EMPTY>
<!ATTLIST extension
   x-min CDATA #REQUIRED
   z-min CDATA #REQUIRED
   x-max CDATA #REQUIRED
   z-max CDATA #REQUIRED
>
```

For example, the model that defines our room can be the coliseum of Rome. This is the XML that defines it:

```
<environment>
   <room>
      <modelRoom uri="assets/model/COLI_L" x="297" y="103" z="5">
         <extension x-max="162" x-min="-250" z-max="180" z-min="-180"/>
         <transformations>
            <objectRotation>
               <rotation-axis-x angle="-90.0"/>
               <rotation-axis-y angle="0.0"/>
               <rotation-axis-z angle="0.0"/>
            </objectRotation>
            <objectScale>
               <scale-axis-x scale="1.975897"/>
               <scale-axis-y scale="1.975897"/>
               <scale-axis-z scale="1.975897"/>
            </objectScale>
         </transformations>
      </modelRoom>
      <sky>
         <north uri="assets/sky/sky_north.jpg"/>
         <south uri="assets/sky/sky_south.jpg"/>
         <west uri="assets/sky/sky_west.jpg"/>
         <east uri="assets/sky/sky_east.jpg"/>
         <top uri="assets/sky/sky_up.jpg"/>
         <bottom uri="assets/sky/sky_down.jpg"/>
      </sky>
   </room>
</environment>
```

Two different point of view of this scene are shown in the next image:



Model rooms are not so cost-reduced as closed rooms since the user need to produce (or hire a 3D model artist) the environment model. Nonetheless, the expressivity gained introducing this kind of environment was very valuable as real environments can be represented in games without developing a full scenario creation tool (which was out of the scope of the project).

Another attribute that we can see in the DTD definition of a scene is the **default initial position.** It is a reference to the player position in the scene when it is loaded. Configuring the initial position we can ensure that the player will not be located onto a wall, or inside an object when there is a change of scene. This is a position in the space so it is defined by three coordinates.

There are two attributes called **objects and characters references** that are two lists that contain the objects (also called items) and characters that appeared in the scene. For each item and character is also set its initial position in the scene. Moreover, transformations can be done to these references. This is very useful because there can be more than one element repeated in the scene with different sizes or rotations. It is also useful to transform the instances instead of doing these changes to the original element because is easier to transform while we are watching the whole scene using the editor tool. We speak about transformations in a more detailed way in section 3.4 of this chapter.

Another part of the definition of the scenes is the attribute **regions**, which is completely optional. When we used closed rooms characters can not walk through the walls because the collisions control system detects the movement into physical beings and impedes the player moves into them. However, in the open environments there are no walls so the player can walk to wherever the user wants. At the first instance this is why we decided to create the concept of region, but now they are useful for many different purposes. At the beginning we define regions as transparent cuboids in the scene where the player can not walk through. Later we extend the concept so now, regions can be walked through by the player or not.

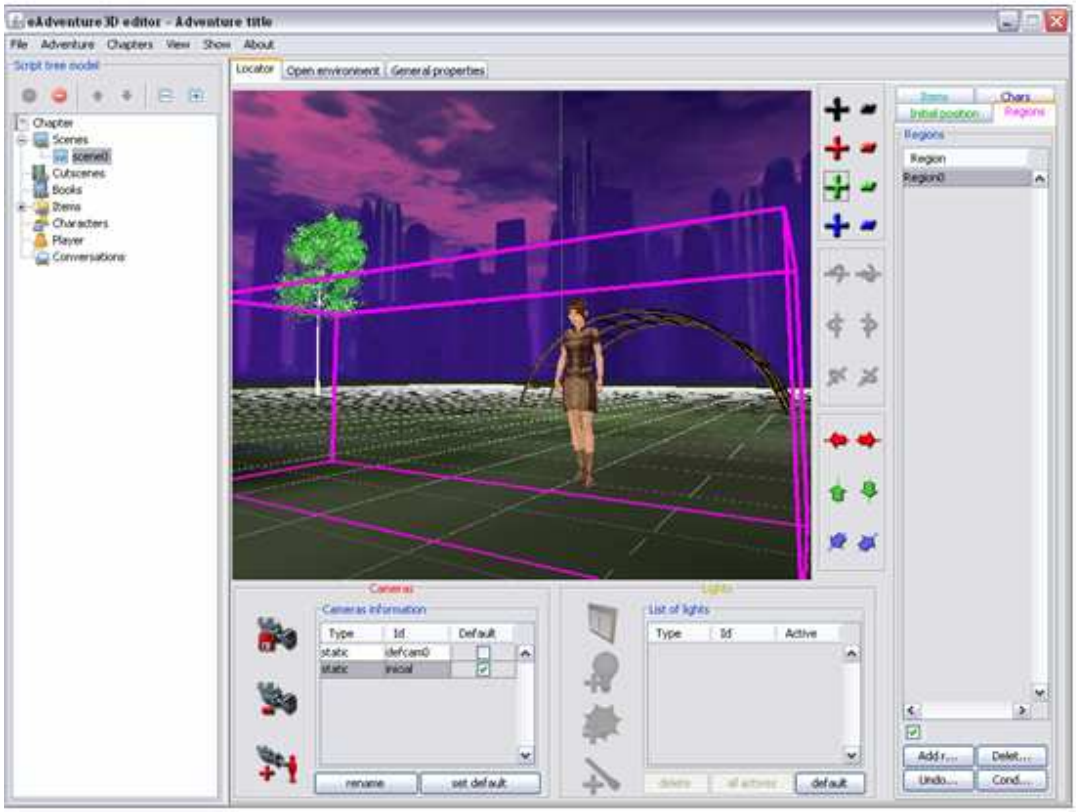
Users can change such behaviour in a region by just selecting or deselecting a simple check box.

If a region can be walked through by the player is useful to us because we can know where the player is and, for example, change the camera or turn on the light (i.e. trigger an effects block). We will see this more detailed in section 3.6 of this chapter speaking about effects. Regions are defined in <e-Adventure3D> like this:

```
<!--REGIONS-->
<!ELEMENT regions (region+)>
<!ELEMENT region (documentation?, condition?, effect?, post-effect?, transformations?)>
<!ATTLIST region
   id CDATA #REQUIRED
   %positionR;
   crossable (yes | no) "yes"
>
```

We are going to see an example based in one of the open environments:

```
<regions>
   <region crossable="no" id="Region0" x="90" y="0" z="228">
      <transformations>
         <objectScale>
            <scale-axis-x scale="1090.7407"/>
            <scale-axis-y scale="394.46466"/>
            <scale-axis-z scale="340.75345"/>
         </objectScale>
      </transformations>
   </region>
</regions>
```



We have added a region to the scene that is not crossable. It won't allow the player to go extremely near to the camera. Regions are pink-drawn in the editor tool so users can modify them, but they are transparent in the engine.

The **view** section of the scene references to cameras and lights. This section is the most complicated part of the language so with the editor tool we tried to simplify it as much as we could. Let's speak about cameras and lights.

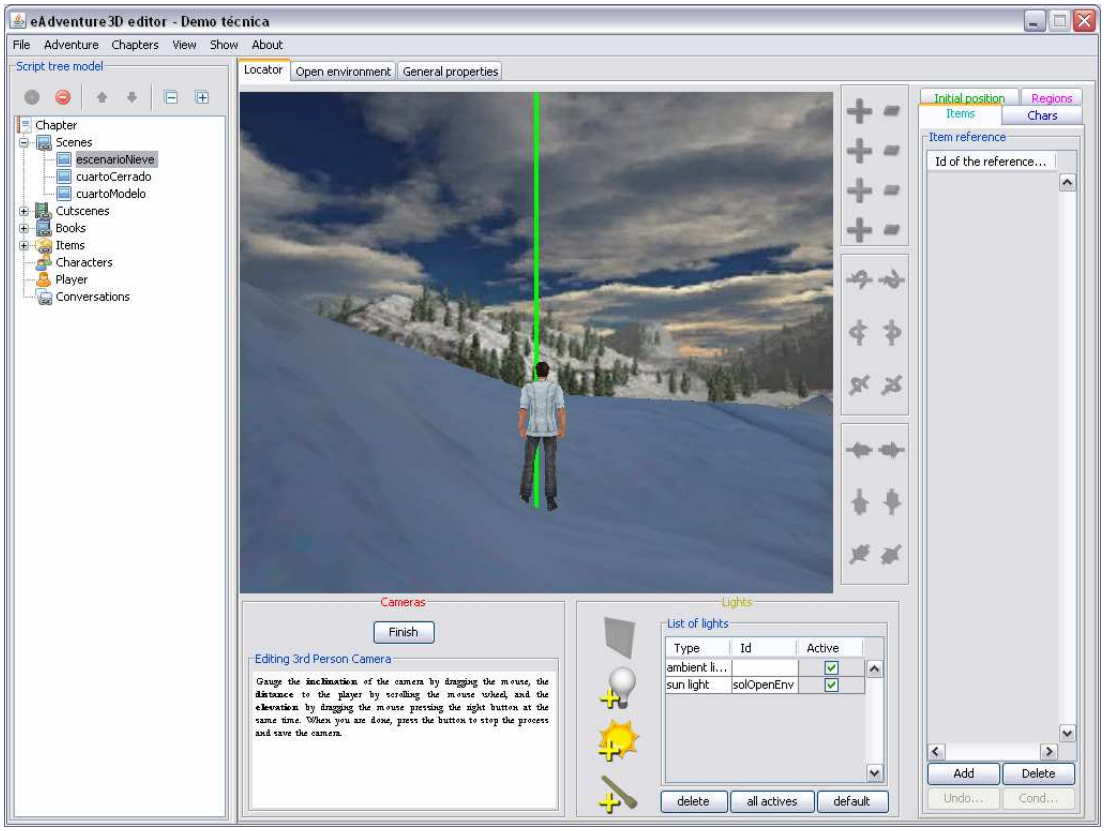
At the beginning of this section we wrote about the difficulty of keeping the simplicity of the <e-Adventure2D> language because extensions like **cameras**. In <e-Adventure2D> the scenes were always seen through a 'static camera' situated in the same place (as it is well known, there is no need for cameras handling in 2D games). In a three dimensional space this approach would be insufficient, so we allow infinites points of view of the same scene.

We allow two kinds of cameras: static and third person cameras. Despite we have simplified as much as we could the parameters that are needed to define a camera, game makers who do not want to use the editor tool must know some of basic geometry. This is the definition of the <e-Adventure3d> language for cameras:

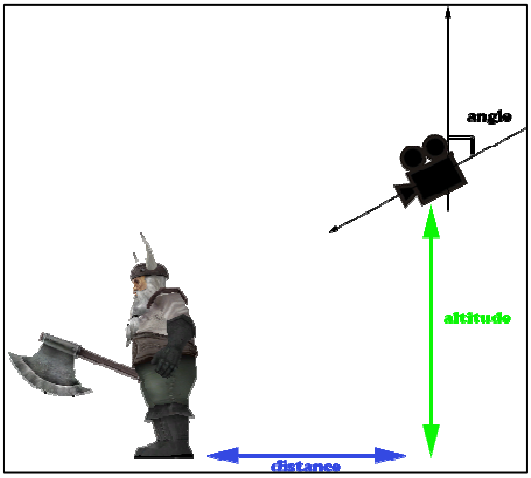
```
<!--CAMERAS-->
<!ELEMENT cameras (staticCamera|thirdPersonCamera)+>

<!--static camera-->
<!ELEMENT staticCamera (direction?, position?, documentation?)>
<!ATTLIST staticCamera
   id ID #REQUIRED
   default-camera (yes|no)  #IMPLIED
>
<!--third person camera-->
<!ELEMENT thirdPersonCamera (cameraData?, documentation?)>
<!ATTLIST thirdPersonCamera
   id ID #REQUIRED
   default-camera (yes|no) #IMPLIED
>
<!ELEMENT cameraData EMPTY>
<!ATTLIST cameraData
   angle NMTOKEN #REQUIRED
   distance NMTOKEN #REQUIRED
   altitude NMTOKEN #REQUIRED
>

<!ELEMENT direction EMPTY>

<!ATTLIST direction
   %positionR;
>
<!ELEMENT position EMPTY>
<!ATTLIST position
   %positionR;
>
```

On the one hand the third person camera is a camera that chases the player along the game. The best way to see how it works is to play a game with this camera but we can see the way of editing them in order to understand the concept:

```
<thirdPersonCamera default-camera="no" id="chaseCamera">
  <cameraData altitude="14.101095" angle="1.4707963" distance="63.69816"/>
</thirdPersonCamera>
```



As the XML extract depicts, some parameters can be gauged on third person cameras for a simple reason: since the player is different from one adventure to another the user must be allowed to adjust how close the camera will be to the player (*distance* attribute). Following the same reasoning player height will be different from one player to other, so the *altitude* of the camera, sized from the floor (or terrain) as a reference must be different in every case. Finally, users can adjust the inclination *angle* in radians (respective to the XZ plane on which all elements are placed) so the view varies as if we were observing the player from different points. The left figure depicts this idea.



On the other hand there are **static cameras** defined by the direction where the camera points to and its position in the space. For simplicity all static cameras are restricted to lay parallel to the XZ plane (roll is not permitted). Such simplification reduces the number of required vectors to define a camera in two instead of three.

We are going to see an example of four different cameras situated in different places of the same scene:

```
<staticCamera default-camera="no" id="camaraTerraza3rd">
   <direction x="0.021410223" y="-0.3292196" z="0.9440106"/>
   <position x="-23.635534" y="117.033875" z="-347.555"/>
</staticCamera>
<staticCamera default-camera="no" id="entradaCamara">
   <direction x="-0.3638072" y="-0.50810176" z="-0.78069013"/>
   <position x="-37.459175" y="179.76706" z="25.949821"/>
</staticCamera>
<staticCamera default-camera="no" id="generalCam">
   <direction x="0.89513266" y="-0.31501454" z="0.3154418"/>
   <position x="-470.13004" y="226.8129" z="-245.78062"/>
</staticCamera>
<staticCamera default-camera="yes" id="inicialCam">
   <direction x="-0.27861503" y="-0.43314517" z="-0.8571808"/>
   <position x="0.45132333" y="78.20823" z="-244.35916"/>
</staticCamera>
```

Definition of static cameras in XML is a bit tricky. Users do not only need to guess right the position where the camera must be located, but also a vector which

defines where the camera points to. It takes too long to guess the right vectors directly in XML, but the editor has simplified this process more that we could have ever imagined before developing the application. As it is usual in 3D edition tools, user can move the view point by just dragging and dropping the mouse. Then the current camera (which defines what the user is seeing in the edition panel) can be automatically saved. During all the process the user did not need to know anything about the coordinates of the camera (i.e. vectors). Actually those coordinates are never given to the user.

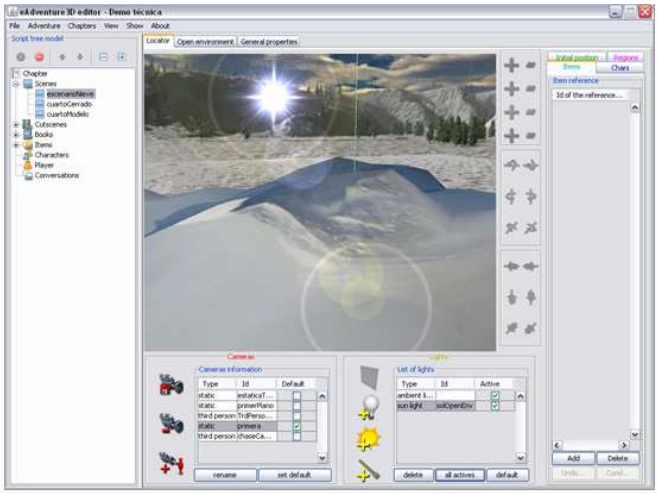
Add **lights** to scenes also require some knowledge of basic geometry. However, it is easier to define default lights for each type of scene than default cameras, so lights are an optional part in <e-Adventure3D> language. There are four different light types: an *ambient light*, a light from a *bulb*, a light from a *lantern* and a light similar to the *sun light*:

```
<!-- LIGHT-->
<!ELEMENT lights (ambient-light?, (bulb-light | lantern-light | sun-light)*)>
<!ELEMENT bulb-light (position,light-color)>
<!ATTLIST bulb-light
    id ID #REQUIRED
    active (yes|no) "yes"
>
<!ELEMENT lantern-light (position,direction,angle,light-color)>
<!ATTLIST lantern-light
    id ID #REQUIRED
    active (yes|no) "yes"
>
<!ELEMENT angle EMPTY>
<!ATTLIST angle
 degrees NMTOKEN #REQUIRED
>
<!ELEMENT sun-light (direction,light-color)>
<!ATTLIST sun-light
    id ID #REQUIRED
    active (yes|no) "yes"
>
<!ELEMENT ambient-light (light-color)>
<!ELEMENT light-color EMPTY>
<!ATTLIST light-color
   red NMTOKEN #REQUIRED
   green NMTOKEN #REQUIRED
   blue NMTOKEN #REQUIRED
   alpha NMTOKEN #IMPLIED
>
```

Ambient light must be combined with other lights, so we will see some examples of the others. In all light types the colour of the light is required.

This is an example of the sun light. To define this light type it is required to define the direction of the rays:

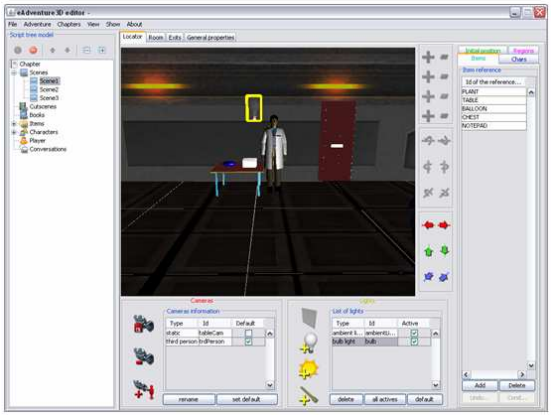
```
<sun-light active="yes" id="solOpenEnv">
    <direction x="-50.0" y="100.0" z="50.0"/>
    <light-color alpha="1.0" blue="1.0" green="1.0" red="1.0"/>
</sun-light>
```

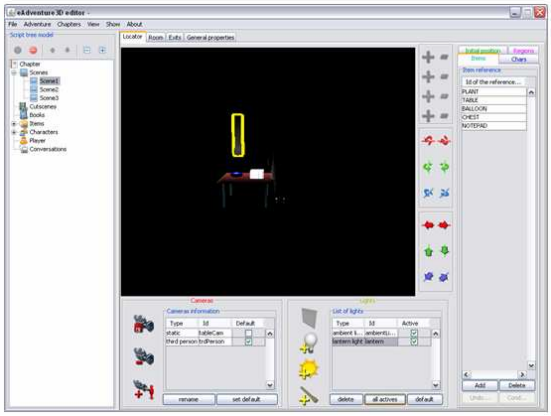


We are going to see the same scene lighted with two different types of lights. In the left image there is a bulb and in the right one we can see a lantern. A bulb is defined by its position and a lantern needs a position, the direction where the light points and the angle covered by the light.

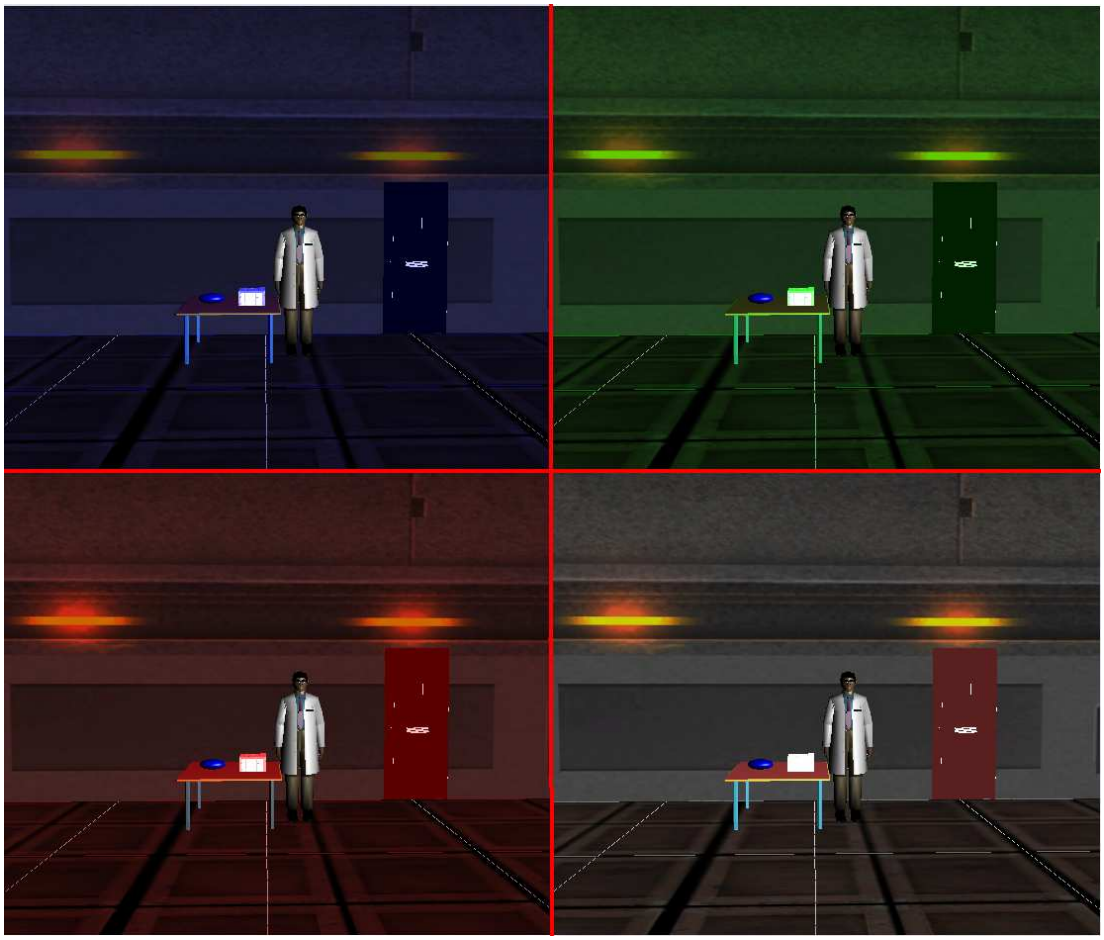
```
<bulb-light active="yes" id="bulb">
    <position x="-25.0" y="24.0" z="-28.0"/>
    <light-color alpha="1.0" blue="1.0" green="1.0" red="1.0"/>
</bulb-light>
```

```
<lantern-light active="yes" id="lantern">
    <position x="-33.0" y="25.0" z="-45.0"/>
    <direction x="0.0" y="-1.0" z="8.940697E-8"/>
    <angle degrees="35.0"/>
    <light-color alpha="1.0" blue="1.0" green="1.0" red="1.0"/>
</lantern-light>
```





Finally, in order to appreciate the effects of the colour in a light we are going to show the same scene lighted by the same bulb with the colour changed:



As a final remark, we will discuss the advantages brought to the platform by the view elements (cameras and lights). On the one hand, combination of cameras and lights enhance the expressivity power of what you can do with <e-Adventure3D>. A good combination of lights and cameras can make the game more attractive from a visual point of view, improving the motivational traits of the game (which is one of the foremost reasons to apply videogames in education).

Nevertheless, the main advantage is not the benefits of a better appearance of the games. The real power of these elements rely on its educational traits. The user can change the current camera and switch on/off lights during the game to guide the user, to give a deeper view of some elements of the game world (i.e. domain of study), or to force the student pay attention on a concrete aspect. Therefore, they are powerful tools to improve the educational value.

3.2 Cut scenes

Secondly we are going to explain the **cut scenes,** the second type of scenes

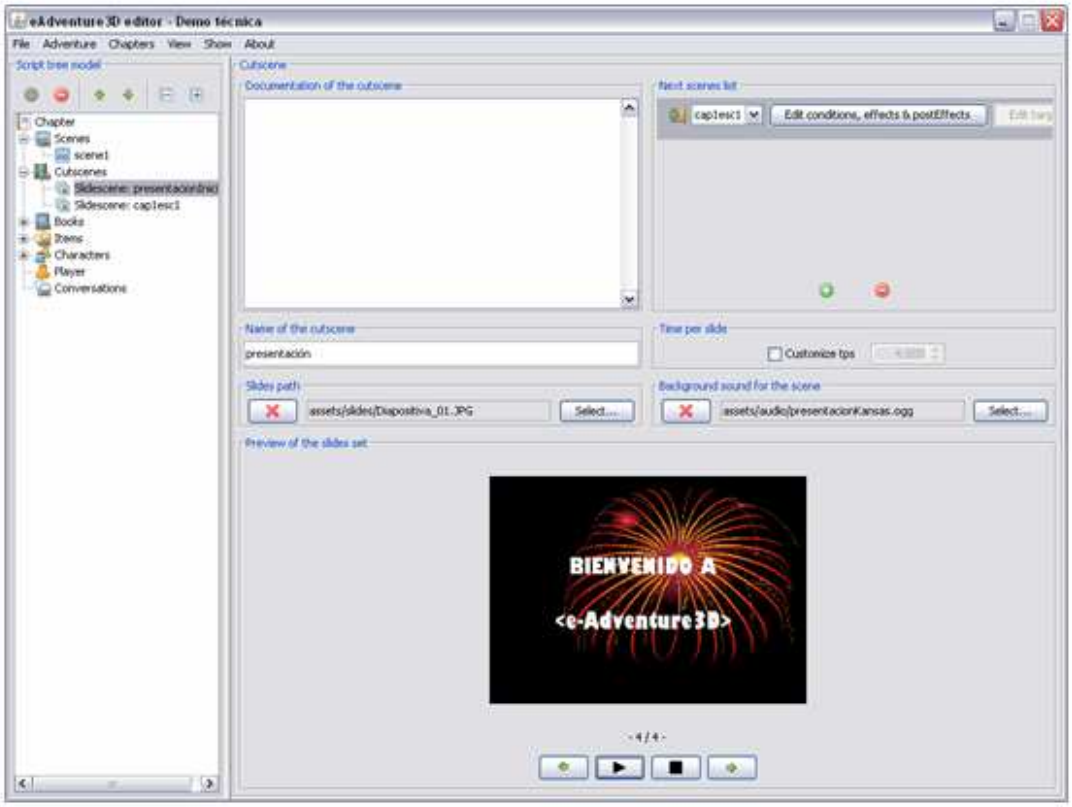
supported by <e-Adventure3D>. These differ from normal scenes in that users cannot interact with them (for that reason they are also known as not-interactive scenes). They are very useful to display information in educational games, and there are two types: *slide scenes* and *video scenes*.

Slide scenes are successions of images (slides) which are full-screen rendered in the screen. To define them users have to enumerate the images that will appear in the scene. Users can also set the time between images and a background sound can be added to the scene as a resource (like in normal scenes).

```
<!--SLIDESCENE-->
<!ELEMENT slidescene (documentation?, resources+, name, end-chapter? ,next-scene*)>
<!ATTLIST slidescene
   id ID #REQUIRED
   start (yes | no) "no"
   time-per-slide NMTOKEN #IMPLIED
>
```

This is an example for a slide scene. As always we can see the XML and the results in the editor tool:

```
<slidescene id="presentacionInicial" start="yes">
   <resources>
      <asset type="bgsound" uri="assets/audio/presentacionKansas.ogg"/>
      <asset type="slides" uri="assets/slides/Diapositiva_01.JPG"/>
   </resources>
   <name>presentación</name>
   <next-scene idTarget="cap1esc1"/>
</slidescene>
```



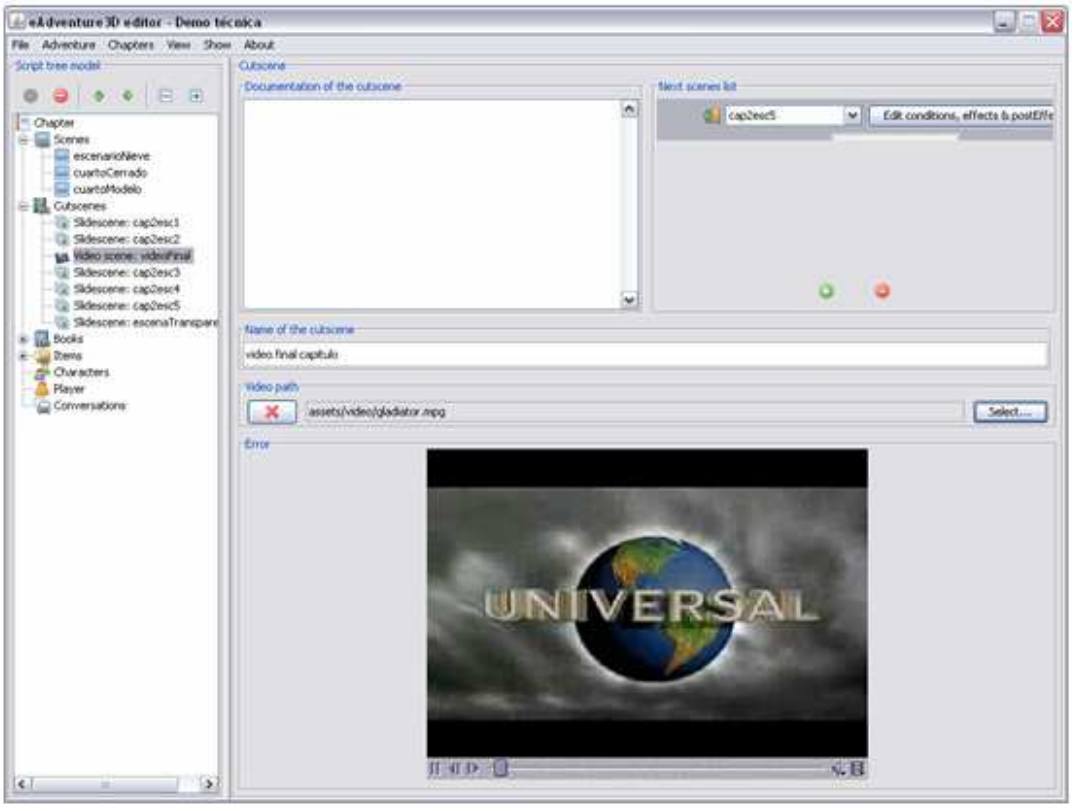
All the slides of the folder ‘slides’ saved in the EA3D file called ‘Diapositiva_’ plus a number will be shown in the cut scene arranged by the number.

Video scenes are just defined by a video. In the engine, the video will be played from the beginning to the end, so a video scene will last the length of the video.

```
<!--VIDEOSCENE-->
<!ELEMENT videoscene (documentation?, resources+, name, end-chapter?, next-scene*)>
<!ATTLIST videoscene
   id ID #REQUIRED
   start (yes | no) "no"
>
```

Next we are going to see an example and the screen used for video scenes edition in the editor tool:

```
<videoscene id="videoFinal" start="no">
   <resources>
      <asset type="video" uri="assets/video/gladiator.mpg"/>
   </resources>
   <name>video final capitulo</name>
   <next-scene idTarget="cap2esc5" x="-1" y="-1" z="-1"/>
</videoscene>
```



AVI, MOV and MPG files are supported. However, please bear in mind that a lot of different encodings can be applied in those videos. For that reason, the correct visualization of a certain video depends on lots of factors such as the operative system, installed codecs, etc., although the video has a valid file extension.

3.3 Books

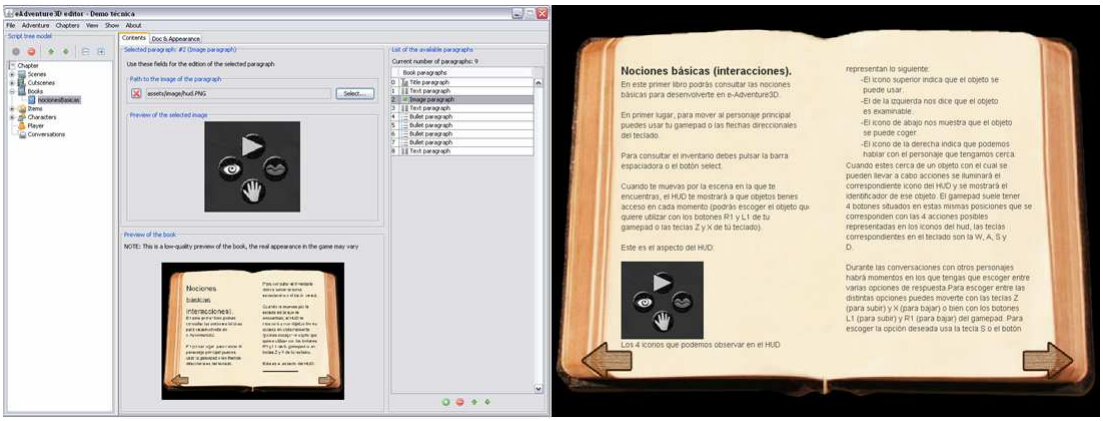
```
<!ELEMENT book (documentation?, resources+, text)>
<!ATTLIST book
   id ID #REQUIRED
>
<!ELEMENT text (#PCDATA | title | bullet | img)*>
<!ELEMENT bullet (#PCDATA)>
<!ELEMENT img EMPTY>
<!ELEMENT title (#PCDATA)>
<!ATTLIST img
   src CDATA #REQUIRED
>
```

Books are elements which are not part of common adventure games. In our case they were devised to improve the educational features of the games, as they can be provided to learners when a lot of information must be transmitted, or just to provide a reference guide of the subject of study.

As we can see in the DTD**, books** have four different types of paragraphs: title, text, bullet and image paragraphs. We must see an example to understand the differences between them:

```
<book id="nocionesBasicas">
   <resources>
      <asset type="image" uri="assets/image/defaultbook.jpg"/>
   </resources>
   <text>
      <title>Nociones básicas (interacciones).</title>En este primer libro podrás consultar las nociones básicas para desenvolverte en e-Adventure3D.

En primer lugar, para mover al personaje principal puedes usar tu gamepad o las flechas direccionales del teclado.

Para consultar el inventario debes pulsar la barra espaciadora o el botón select.

Cuando te muevas por la escena en la que te encuentras, el HUD te mostrará a que objetos tienes acceso en cada momento (podrás escoger el objeto que quiere utilizar con los botones R1 y L1 de tu gamepad o las teclas Z y X de tú teclado).

Este es el aspecto del HUD:
      <img src="assets/image/hud.PNG"/>Los 4 iconos que podemos observar en el HUD representan lo siguiente:
      <bullet>El icono superior indica que el objeto se puede usar.</bullet>
      <bullet>El de la izquierda nos dice que el objeto es examinable.</bullet>
      <bullet>El icono de abajo nos muestra que el objeto se puede coger.</bullet>
      <bullet>El icono de la derecha indica que podemos hablar con el personaje que tengamos cerca.</bullet>Cuando estes cerca de un objeto con el cual se pueden llevar a cabo acciones se iluminará el correspondiente icono del HUD y se mostrará el identificador de ese objeto. El gamepad suele tener 4 botones situados en estas mismas posiciones que se corresponden con las 4 acciones posibles representadas en los iconos del hud, las teclas correspondientes en el teclado son la W, A, S y D.

Durante las conversaciones con otros personajes habrá momentos en los que tengas que escoger entre varias opciones de respuesta.Para escoger entre las distintas opciones puedes moverte con las teclas Z (para subir) y X (para bajar) o bien con los botones L1 (para subir) y R1 (para bajar) del gamepad. Para escoger la opción deseada usa la tecla S o el botón inferior de los 4 encionados en el gamepad.
   </text>
```

The next two screen shots show the results in the editor (left) and the view in the engine (right):



3.4 Element definitions: Players, characters and items

Some small differences arise at the time of defining characters, players and objects as we can see in the DTD definition of each of them:

```
<!--OBJECT (ITEM)-->
<!ELEMENT object (documentation?, instance*, resources+, description, actions?, transformations?)>
<!ATTLIST object
   id ID #IMPLIED
>
<!--PLAYER-->
<!ELEMENT player (documentation?, resources+, textcolor?, description, characterDirection, characterUpVector, characterAnimations?, transformations?)>

<!--CHARACTER-->
<!ELEMENT character (documentation?, resources+, textcolor?, description, conversations?, characterDirection?,characterUpVector?,characterAnimations?,
transformations?)>
```

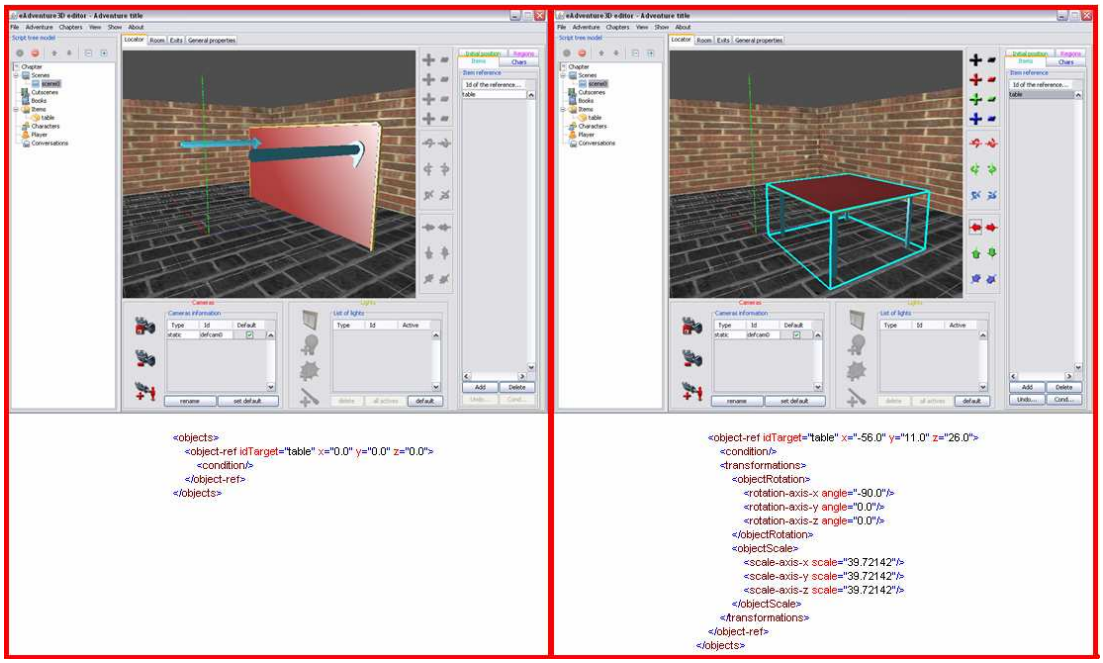
We are going to explain the characteristics they share:

The **resources** block contains the references to the art files used in the renderization of the element. These must include the following assets paths: the one where we saved the 3D model that represents the element, the path where the texture we want to apply to the model is and, speaking about items, the URI where the icon that represents the item is saved (just in case the player can grab it, so that it can be represented in the inventory). JME allows 3d models designed with different programs, the extensions admitted are: 3DS, ASE, DAE, JME, MD2, MD3, MS3D and OBJ.

3D models to be loaded in games should be created according to the scene so they could fix in it, but in case they do not, we have defined some transformations that can be applied to them. The scale **transformations** change the size of the models and the rotation transformations turn the models over the three different axes. This is how transformations are defined in the DTD file:

```
<!--TRANSFORMATIONS-->
<!ELEMENT transformations ((objectRotation | objectScale)+)>
<!ELEMENT objectScale (scale-axis-x?,scale-axis-y?,scale-axis-z?)>
<!ELEMENT objectRotation (rotation-axis-x?, rotation-axis-y?, rotation-axis-z?)>
<!ELEMENT rotation-axis-x EMPTY>
<!ELEMENT rotation-axis-y EMPTY>
<!ELEMENT rotation-axis-z EMPTY>
<!ATTLIST rotation-axis-x
   %angle;
>
<!ATTLIST rotation-axis-y
   %angle;
>
<!ATTLIST rotation-axis-z
   %angle;
>

<!ATTLIST scale-axis-x
   scale NMTOKEN #REQUIRED
>
<!ATTLIST scale-axis-y
   scale NMTOKEN #REQUIRED
>
<!ATTLIST scale-axis-z
   scale NMTOKEN #REQUIRED
>
<!ELEMENT scale-axis-x EMPTY>
<!ELEMENT scale-axis-y EMPTY>
<!ELEMENT scale-axis-z EMPTY>
```

When it is necessary transformations can be applied to 3D models. Transformations are very useful when someone uses models that had not been designed for his own project, or just to reuse the same models in different situations. In the example adventure we used free models from the Internet that did not fix from the beginning in our scenes, but we applied transformations to the models. After the transformation models fixed in the scenes and it seemed like we had designed them for our example adventure. Let's see an example:



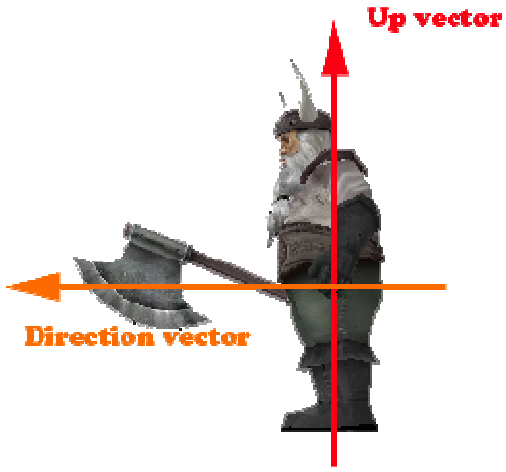
In the left image we have just loaded an item which is a table. We have downloaded it from the Internet so it has not been designed for this game and, as we can see, it does not fix well. The right image is the result after applying some transformations to the item reference in the scene: it has been scaled, so that now its size has been reduce to the 39%; it has been rotated ninety degrees by the x axis and we have also changed its position in the scene. Although this process was extremely arduous when it was required to be written by hand, users must not worry as the editor simplifies extremely this operation.

Characters and the player have four extra attributes:

- **Text colour**: this attribute must be specified if we are going to use conversations with subtitles. Defines subtitles colour for this character or player, useful to distinguish which character is talking in multiple-character conversations.
- **Character up vector** defines the axis along which the “spine” of the character lays, oriented to the head.
- **Character direction** is a vector which tells the engine where the character is facing. This vector must be normal to the “face” of the character.

  In combination with the up vector, both vectors give the engine the information required for the movement of the player and for the definition of third person cameras. Without them it could happen that the

character moves "backwards" all the time, which obviously is not a desired behaviour although the character is Michael Jackson.

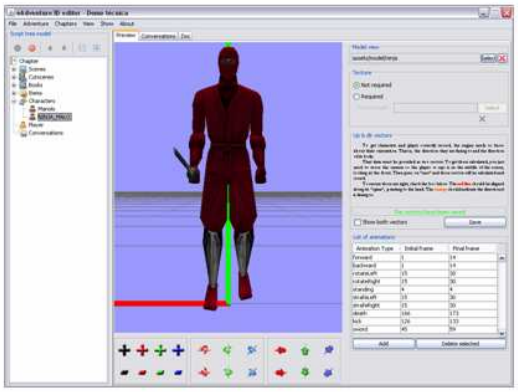


- Some 3D models have their **animations** associated to different frame ranges. Hence, users have to specify which the initial frame and the final frame are for each animation, and give a name for it so it could be easily referred.

```
<!ELEMENT animation EMPTY>
<!ELEMENT characterAnimations (animation+)>
<!ATTLIST animation
   name CDATA #REQUIRED
   initialFrame NMTOKEN #REQUIRED
   finalFrame NMTOKEN #REQUIRED
>
```

Characters also have the attribute called conversations; it keeps references to the conversations between the character and the player.

Let's see an example of a character definition:



```
<character id="NINJA_MALO">
  <documentation/>
  <resources>
    <asset type="model" uri="assets/model/ninja"/>
  </resources>
  <textcolor>
    <frontcolor color="#ff0000"/>
    <bordercolor color="#000000"/>
  </textcolor>
  <description>
    <name>Ninja</name>
    <brief/>
    <detailed/>
  </description>
  <characterDirection x="0.0" y="0.0" z="-1.0"/>
  <characterUpVector x="0.0" y="1.0" z="0.0"/>
  <characterAnimations>
    <animation finalFrame="14" initialFrame="1" name="forward"/>
    <animation finalFrame="14" initialFrame="1" name="backward"/>
    <animation finalFrame="30" initialFrame="15" name="rotateLeft"/>
    <animation finalFrame="30" initialFrame="15" name="rotateRight"/>
    <animation finalFrame="4" initialFrame="4" name="standing"/>
    <animation finalFrame="30" initialFrame="15" name="strafeLeft"/>
    <animation finalFrame="30" initialFrame="15" name="strafeRight"/>
    <animation finalFrame="173" initialFrame="166" name="death"/>
    <animation finalFrame="133" initialFrame="126" name="kick"/>
    <animation finalFrame="59" initialFrame="45" name="sword"/>
  </characterAnimations>
  <transformations>
    <objectScale>
      <scale-axis-x scale="219.99988"/>
      <scale-axis-y scale="219.99988"/>
      <scale-axis-z scale="219.99988"/>
    </objectScale>
  </transformations>
</character>
```

The left part of the image shows how the XML (in the right part) is represented in the editor tool. There are two more tabs, one with the list of conversations and the other is reserved for the documentation and the text colour.

In reference to the items there is one attribute called **actions**; it keeps the list of actions that can be done with the item we are specifying. There are four possible actions:

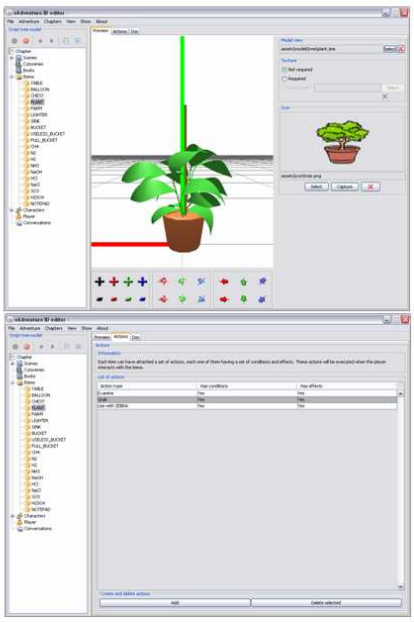
- Grab (south icon of the HUD): Indicates that the item can be grabbed.
- Examine (west icon of the HUD): The item can be examined.
- Use (north icon of the HUD): The item can be used.
- Use with (north icon of the HUD): ones the item is kept in the player inventory, it can be used with other objects or characters.

These actions can be done with an item selected in the HUD (when the player is near to this item). Lets remember the HUD now that we have explained the possible actions:

Actions provides the interaction in <e-Adventure3D>. The available actions for a certain object in a certain moment is subjected to the satisfaction of some conditions, as it will be explained later, and permits the execution of a block of effects (where the interaction is defined).

The icon in the east part of the HUD is reserved to characters and indicates if a character has conversations available.

Next we can see an example of an item edition in the editor and the XML generated:



```
<object id="PLANT">
  <documentation/>
  <resources>
    <asset type="model" uri="assets\model\tree\plant.jme"/>
    <asset type="icon" uri="assets\icon\tree.png"/>
  </resources>
  <description>
    <name>The amazing tree</name>
    <brief>?</brief>
    <detailed>?</detailed>
  </description>
  <actions>
    <examine>
      <condition>
        <active flag="scene1Active"/>
      </condition>
      <effect>
        <trigger-conversation idTarget="examinePlant1"/>
      </effect>
    </examine>
    <grab>
      <condition>
        <active flag="scene1Active"/>
      </condition>
      <effect>
        <generate-object idTarget="PLANT"/>
        <activate flag="plantGrabbed"/>
      </effect>
    </grab>
    <use-with idTarget="ZEBRA">
      <condition>
        <active flag="usedBalloonWithZebra"/>
      </condition>
      <effect>
        <generate-object idTarget="PLANT"/>
        <activate flag="plantGrabbed"/>
      </effect>
    </grab>
    <use-with idTarget="ZEBRA">
      <condition>
        <active flag="usedBalloonWithZebra"/>
      </condition>
      <effect>
        <consume-object idTarget="PLANT"/>
        <activate flag="usedPlantWithZebra"/>
        <trigger-conversation idTarget="plantUsedWithZebra"/>
        <change-camera idTarget="zebraBottom"/>
        <trigger-conversation idTarget="balloonInflated"/>
        <change-camera idTarget="scopeCamera"/>
        <generate-object idTarget="CH4"/>
      </effect>
    </use-with>
  </actions>
  <transformations>
    <objectRotation>
      <rotation-axis-x angle="270.0"/>
      <rotation-axis-y angle="0.0"/>
      <rotation-axis-z angle="0.0"/>
    </objectRotation>
    <objectScale>
      <scale-axis-x scale="10.0"/>
      <scale-axis-y scale="10.0"/>
      <scale-axis-z scale="10.0"/>
    </objectScale>
  </transformations>
</object>
```

In the fist tab of the editor we have a preview window with the model (in the first image). We also can see the icon associated with it. The second tab (in the second image) contains the actions associated to this item. There is another tab that contains the documentation.

3.5. Conversations

Conversations are very important in graphical adventures, as they provide both interaction with characters and a good source of guidance for the player. Those aspects are therefore very interesting for educational games, as they can be used to guide the student during the learning experience and provide information (they are an entertaining way of explaining things to the student).

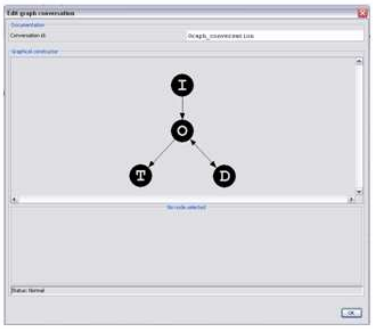
There are two types of conversations in <e-Adventure3D>: graph conversations and tree conversations. Both are very similar, differing just in the possible paths the conversation can take in each case.

Firstly, let's see the DTD definition part for the two types:

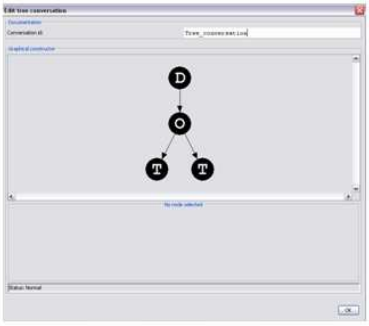
```
<!--CONVERSATION-->

<!ENTITY % conversation "tree-conversation | graph-conversation">
<!ENTITY % continuation "(response|end-conversation)">

<!--TREE CONVERSATION-->
<!ELEMENT tree-conversation (%speak;, %continuation;)>
<!ATTLIST tree-conversation
   id ID #REQUIRED
>
<!--GRAPH CONVERSATION-->
<!ELEMENT graph-conversation (dialogue-node | option-node)+>
<!ATTLIST graph-conversation
   id ID #REQUIRED
>
<!ELEMENT dialogue-node ((effect?,sound?,(speak-player | speak-npc))*, (child | end-conversation))>
<!ATTLIST dialogue-node
   nodeindex CDATA #REQUIRED
>
<!ELEMENT option-node (effect?,sound?,speak-player , child)+>
<!ATTLIST option-node
   nodeindex CDATA #REQUIRED
>
<!ELEMENT child EMPTY>
<!ATTLIST child
   nodeindex CDATA #REQUIRED
>
<!ELEMENT sound EMPTY>
<!ATTLIST sound
   url CDATA #REQUIRED
>
<!ELEMENT response (option)+>
<!ELEMENT option (effect?, sound?,speak-player, %speak;, (%continuation; | go-back))>
<!ELEMENT go-back EMPTY>
<!ELEMENT end-conversation (effect?)>
```

Secondly, this is an example where we can see an XML of the two conversation types and their editions in the editor tool:

```
<graph-conversation id="Graph_conversation">
   <dialogue-node nodeindex="0">
      <speak-player>Hi, how are you?</speak-player>
      <child nodeindex="1"/>
   </dialogue-node>
   <option-node nodeindex="1">
      <speak-player>I'm OK</speak-player>
      <child nodeindex="2"/>
      <speak-player>I'm not OK</speak-player>
      <child nodeindex="3"/>
   </option-node>
   <dialogue-node nodeindex="2">
      <speak-npc idTarget="TEACHER">I'm glad to hear that</speak-npc>
      <end-conversation/>
   </dialogue-node>
   <dialogue-node nodeindex="3">
      <speak-player>But I have good news for you, you have passed all your exams. How are you now?</speak-player>
      <child nodeindex="1"/>
   </dialogue-node>
</graph-conversation>
```



```
<tree-conversation id="Tree_conversation">
   <speak-player>Hi!</speak-player>
   <speak-npc idTarget="TEACHER">Hi!</speak-npc>
   <speak-npc idTarget="TEACHER">Do you want to start the class?</speak-npc>
   <response>
      <option>
         <speak-player>No.</speak-player>
         <speak-npc idTarget="TEACHER">Then go home</speak-npc>
         <end-conversation/>
      </option>
      <option>
         <speak-player>I'm not fine today.</speak-player>
         <speak-player>Then go home</speak-player>
         <end-conversation/>
      </option>
   </response>
</tree-conversation>
```



Conversations are defined in terms of *nodes*. Although the idea can be tricky for instructors (i.e. game makers) with no programming skills, this conception is necessary to support conversations in which the user participates actively by choosing, at some points of the conversation, between some options which is the next dialogue line the player must say. This is a common feature in classic adventure games, and without it conversations would be monotonous, boring and not useful for educational purposes.

In this manner, there are two basic types of nodes: *Dialogue and option nodes*. The first type is devised to contain the dialogue lines of the conversation. When the user is desired to choose between options, a new option node must be created and linked to the previous dialogue node. Then, each option defined in the node will lead to a different dialogue node, driving the conversation to a different path in each case.

Mainly, the difference between the two types of conversations is that graph conversations allow cycles while the tree conversations not (as we can see in the image). Tree conversations have an easier syntax because they are always linear. However, all tree conversations can be written as graph conversations.

3.6 Flags, conditions and effects

Flags are booleans that determinate the game state in each moment. For people with no programming knowledge, just think of flags as an indicator which can be "active" or "inactive" at any moment, as traffic lights can be green or red. In fact flags can be compared to traffic lights as they control the narrative flow of the games, allowing the traffic to pass only in the second case.

As booleans, flags can be activated or deactivated and depending on their state the game will take one direction or other. So that flags are used by game makers to establish the game course.

The question is: how can game makers use flags? Flags are used through the **conditions**. Conditions can be applied in different parts of a game:

- Next scenes: before the player can go to another scene, there can be some conditions that should be verified.
- Regions: in crossable regions there can be conditions that should be verified before region effects can be launched. (Effects are explained

later).

- Actions and conversations: sometimes is very useful to use conditions with the actions associated with an item. The same thing happens with conversation references associated with a character. In this manner we can define several actions or conversations of the same type, and the engine will pick the first which conditions are satisfied, supporting different behaviours along the game moves on.
- References: items and characters references in a scene can have conditions, so that these elements will appear in the scene when their conditions are verified, or disappear when they are not verified.

Conditions are defined like that in <e-Adventure3D> language:

```
<!--CONDITION-->
<!ENTITY % basic-condition "(active|inactive)">
<!ELEMENT condition (%basic-condition; | either)*>
<!ELEMENT active EMPTY>
<!ATTLIST active
   flag NMTOKEN #REQUIRED
>
<!ELEMENT inactive EMPTY>
<!ATTLIST inactive
   flag NMTOKEN #REQUIRED
>
<!ELEMENT either (%basic-condition;)+>
```

And, how can we change the flags? Flags can be changed with the **effects**. So that, effects are automatic actions that can be launch in many parts of a game:

- Next scenes: When the scene changes some effects could be launched in the current scene (normal effects) and some other effects could be launched in the next scene (post effects).
- Regions: When the player walks into a region some effects could be launched at the entrance (normal effects) and some others could be launched when the player leaves the region (post effects).
- Actions: Some effects could be launched after an action associated with an item
- Conversations: During a conversation, effects could be launched after each line (and when effects end the current conversation will continue normally) and also at the end of the conversation.

Effects are mostly used in the same places as conditions. Mostly because some effects are launched after verifying that some conditions (so the game state) are satisfied.

As we can see in the DTD, effects are not used only to change the flags values:

```
<!--EFFECT-->
<!ENTITY % effects "(activate | deactivate | consume-object | generate-object | play-sound | cancel-action | speak-player | speak-npc | set-player-animation |
set-npc-animation | move-player | move-npc | change-light | change-camera | trigger-conversation | trigger-cutscene | trigger-next-scene | trigger-book |
trigger-end-chapter)*">
<!ELEMENT effect (%effects;)>
<!ELEMENT post-effect (%effects;)>
<!ELEMENT activate EMPTY>
<!ATTLIST activate
   flag NMTOKEN #REQUIRED
>
<!ELEMENT deactivate EMPTY>
<!ATTLIST deactivate
   flag NMTOKEN #REQUIRED
>
<!ELEMENT consume-object EMPTY>
<!ATTLIST consume-object
   idTarget IDREF #REQUIRED
>
<!ELEMENT generate-object EMPTY>
<!ATTLIST generate-object
   idTarget IDREF #REQUIRED
>
<!ELEMENT play-sound EMPTY>
<!ATTLIST play-sound
   background (yes | no) "yes"
   uri CDATA #REQUIRED
>
<!ELEMENT cancel-action EMPTY>

<!ELEMENT speak-player (#PCDATA)>
<!ATTLIST speak-player
   uri CDATA #IMPLIED
>
<!ELEMENT speak-npc (#PCDATA)>
<!ATTLIST speak-npc
   idTarget IDREF #REQUIRED
   uri CDATA #IMPLIED
>
<!ELEMENT set-player-animation EMPTY>
<!ATTLIST set-player-animation
   animationId CDATA #REQUIRED
   repeat (yes | no) "no"
>
<!ELEMENT set-npc-animation EMPTY>
<!ATTLIST set-npc-animation
   characterId IDREF #REQUIRED
   animationId CDATA #REQUIRED
   repeat (yes | no) "no"
>

<!ELEMENT move-player (target-point+)>
<!ELEMENT move-npc (target-point+)>
<!ATTLIST move-npc
   idTarget IDREF #REQUIRED
>
<!ELEMENT target-point EMPTY>
<!ATTLIST target-point
   x CDATA #REQUIRED
   z CDATA #REQUIRED
>

<!ELEMENT change-light EMPTY>
<!ATTLIST change-light
   idTarget IDREF #REQUIRED
>
<!ELEMENT change-camera EMPTY>
<!ATTLIST change-camera
   idTarget IDREF #REQUIRED
>
<!ELEMENT trigger-conversation EMPTY>
<!ATTLIST trigger-conversation
   idTarget IDREF #REQUIRED
>

<!ELEMENT trigger-cutscene EMPTY>
<!ATTLIST trigger-cutscene
   idTarget IDREF #REQUIRED
>

<!ELEMENT trigger-next-scene EMPTY>
<!ATTLIST trigger-next-scene
   idTarget IDREF #REQUIRED
>

<!ELEMENT trigger-book EMPTY>
<!ATTLIST trigger-book
   idTarget IDREF #REQUIRED
>

<!ELEMENT trigger-end-chapter EMPTY>
```

The next table summarizes the different effects types, the effect parameters that are needed to create it and the function of the effect:

Name of the effect	Parameters	Function
Activate	A flag id	Activates the flag.
Deactivate	A flag id	Deactivates the flag.
Consume object	The item id	Quits the item from the inventory.
Generate object	The item id	Generates the item in the inventory.
Play sound	The uri that indicates where the sound is saved and the background attribute that can be set as yes (if we want the sound to iterate) or not (if we want to play the sound only this time)	Plays the sound. It will sound louder if it is not a background sound.
Speak player	The uri that indicates where the sound is saved. The sentence written between the XML marks.	This effect writes the sentences as a subtitle and plays the sound at the same time.
Speak character	Same parameters as speak player plus the id of the character.	The same as in speak player effect, but the subtitle will be written with the font colour associated with this character.
Set player animation	The animation id and the attribute that indicates if the animation must repeat one time or repeat until it is changed.	The player will do the corresponding animation.
Set character animation	The same that in the previous effect plus the character id.	The character will do the corresponding animation.

Move player	The list of points (only the coordinate X and Z) the player must visit.	The player will walk through all the points listed, dodging the items and characters in the trajectory.
Move character	The same parameters of move player plus the character id.	The character will walk through all the points listed, dodging the items and other characters in the trajectory.
Change light	The light id	If the light referenced is on, it will be turned off. If the light referenced is off, it will be turned on.
Change camera	The camera id	Changes the current camera to the camera referenced. So it changes the point of view of the current scene.
Trigger conversation	The conversation id	Launches the conversation referenced.
Trigger next scene	The scene id.	Changes the current scene to the scene referenced. It also changes the camera to the default camera of the next scene.
Trigger cut scene	The video scene or the slide scene id.	Launches the cut scene referenced.
Trigger book	The book id.	Shows the book referenced.
Trigger end chapter		Changes to the next chapter according to the descriptor of the adventure (it changes to the initial scene of the new chapter with the default camera of the scene). If there are no more chapters the game ends.

3.7 Assessment and adaptation

With all the features explained so far users can produce complete and powerful adventure games. However their educational value would be limited and it is an aspect which cannot be left aside. Books, cut-scenes, conversations, actions, cameras and lights can improve the educational value of the games but instructors need to know if the learning experience is successful (i.e. the learner achieves the educational goals proposed by the instructor). In other case instructors may be obliged to rely on pre and post-tests before and after the execution of the game. That is what is usually named as *assessment*.

On the other hand, not all the students show the same abilities nor have an egalitarian knowledge of the subject in study. For that reason, it is advisable to adapt the behaviour of the game depending on who is playing; that is what we call *adaptation*.

3.7.1. Assessment

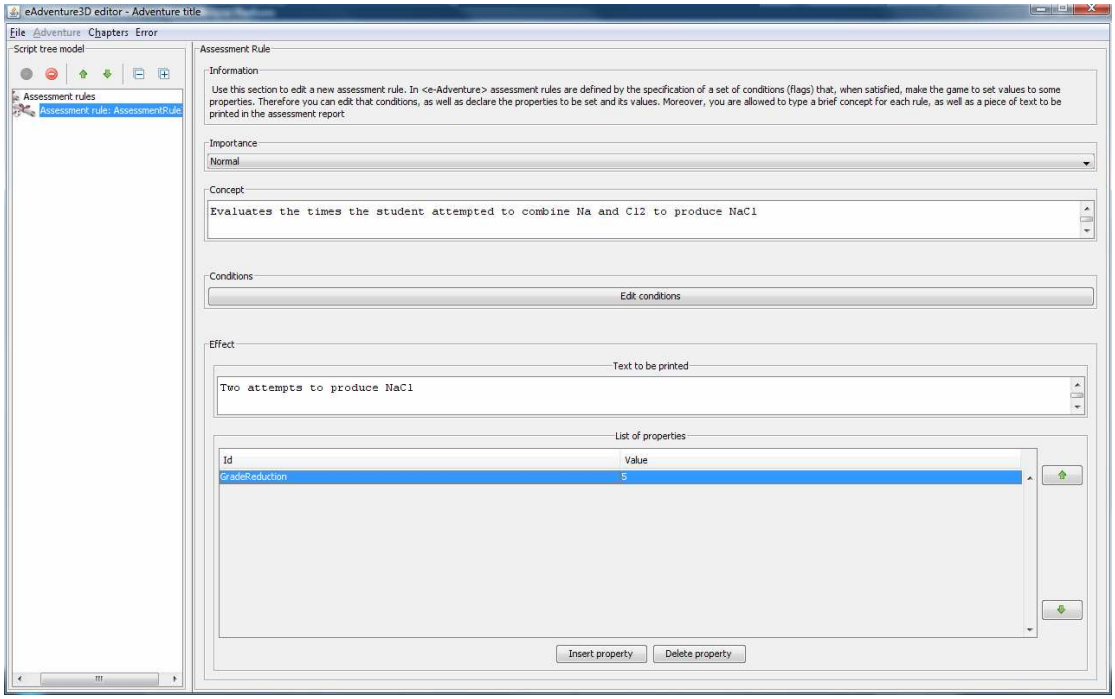
One of the key points of the project is the possibility to define *assessment rules* for the automatic evaluation of the student. The instructor can take advantage of the close monitoring that can be carried out in the game to produce a report with final or partial marks for the student. In addition, we can deliver this report along with a set of variables which have been given a certain value to a LMS (Learning Management System), so the evaluation could be attached to the student.

The set of assessment rules (assessment profile) varies from one chapter to other. Each profile is stored in a different XML document and copied to the EA3D file of the adventure. The following is the complete DTD for the assessment profiles:

```
<!ELEMENT assessment-rules (assessment-rule*)>
<!ELEMENT assessment-rule (concept, condition, effect)>
<!ATTLIST assessment-rule
   id ID #REQUIRED
   importance ( verylow | low | normal | high | veryhigh ) #REQUIRED
>

<!ELEMENT concept (#PCDATA)>

<!ENTITY % basic-condition "(active|inactive)">
<!ELEMENT condition (%basic-condition; | either)+>
<!ELEMENT active EMPTY>
<!ATTLIST active
   flag NMTOKEN #REQUIRED
>
<!ELEMENT inactive EMPTY>
<!ATTLIST inactive
   flag NMTOKEN #REQUIRED
>
<!ELEMENT either (%basic-condition;)+>

<!ELEMENT effect (set-text?, set-property*)>
<!ELEMENT set-text (#PCDATA)>
<!ELEMENT set-property EMPTY>
<!ATTLIST set-property
   id NMTOKEN #REQUIRED
   value NMTOKEN #REQUIRED
>
```

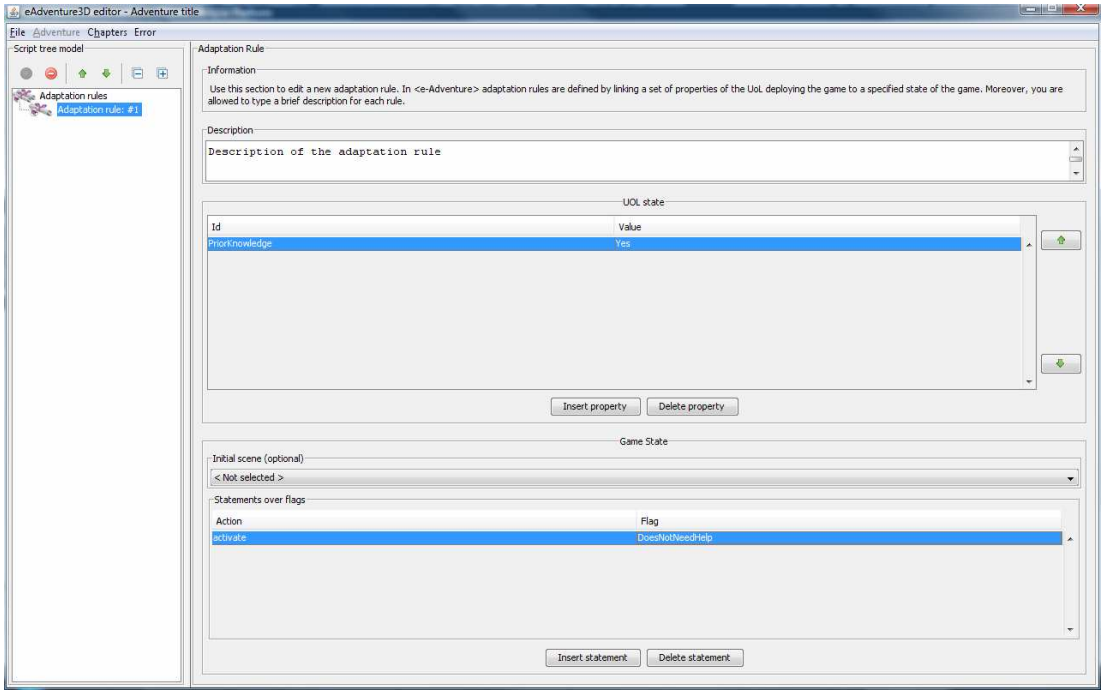
Assessment rules are simple to define. As the DTD depicts, each rule is defined by a group of conditions (in terms of flags) and the effect to be triggered when those conditions are verified. The effect is determined by an optional text to be printed on the report when the rule is executed, and a set of pairs (variable, value), which are the properties to be delivered to the LMS.

The next figure shows how easy is the creation of assessment rules with the editor:



3.7.2. Adaptation

In a very similar manner works adaptation. We can define adaptation rules in which we bind a state of the Unit of Learning (delivered from the LMS as pairs <property, value>) to a state of the game (in terms of activation and deactivation of flags). Besides, a different initial scene can be defined. This can be used, for instance, to skip some first levels for those students with prior knowledge.



In this manner <e-Adventure3D> games are not “black boxes” as opposite to commercial games: we use the rich player-game interaction produced for

assessment and adaptation purposes.

```
<!ELEMENT adaptation (initial-state?, adaptation-rule*)>
<!ELEMENT initial-state (initial-scene?, (activate | deactivate)*)>
<!ELEMENT adaptation-rule (description, uol-state, game-state)>
<!ELEMENT description (#PCDATA)>
<!ELEMENT uol-state (property*)>
<!ELEMENT property EMPTY>
<!ATTLIST property
   id NMTOKEN #REQUIRED
   value NMTOKEN #REQUIRED
>
<!ELEMENT game-state (initial-scene?, (activate | deactivate)*)>
<!ELEMENT initial-scene EMPTY>
<!ATTLIST initial-scene
   idTarget NMTOKEN #REQUIRED
>
<!ELEMENT activate EMPTY>
<!ATTLIST activate
   flag NMTOKEN #REQUIRED
>
<!ELEMENT deactivate EMPTY>
<!ATTLIST deactivate
   flag NMTOKEN #REQUIRED
>
```

3.8 Conclusions

After everything we have told stands out that there are many differences between both projects (<eAdventure3D> and <eAdventure2D>). At the first sight, it could seem that defining an adventure in <eAdventure-3D> could be more complex than in the previous version. However, in our opinion we have found the way to create graphical 3D adventures easily with amazing results.

On the one hand, if users want to write the storyboard of their adventures (which is completely unnecessary because the editor tool) they would find it very easy except for cameras (that requires to know some geometry basics) and transformations. However, if game makers have the appropriate models they will not have to use the transformations so they do not have to know anything about scales or rotations. Finally if they use predefined environments it is more or less as easier as it was in <eAdventure-2D>, but they will have to choose a texture or the sky images instead of a background image.

On the other hand, the editor tool is very powerful and everything that can be written by hand can be developed with the editor where there are no problems with cameras, lights or transformations and the assets are saved automatically.

The most important fact is that without more effort that in <e-Adventure2D>

more complex adventures will be developed and these adventures will gain a lot on realism.

As a final consideration, just remark the high educational value of the adventures produced with the platform. We have not only included elements especially devoted for education, such as cut-scenes or books, but also the rest of the features present interesting applications to improve the educational value. Cameras and lights, for instance, can be used to promote attention and motivation, and to get the student focused on relevant aspects of the domain of study when it is appropriate.

And on top of these features, the assessment and adaptation layers allow instructors to gauge the behaviour of the games according to the student and get an automatic evaluation of the learning experience. Moreover, these data can be stored directly to the profile of the student in a LMS, which cooperation is very interesting if we want educational games to take a place in the curricula and not be relegated to complementary material to be used from time to time.

Chapter X

Development of the project

1. Main Goals

This section explains the expectations of the project divided in iterations and the final goals we have achieved in each of them:

1.1. Preliminary iteration

There was a preliminary phase which its main goal was to confirm if the project we were supposed to develop was possible to be developed. Firstly we chose the technologies (mainly JME engine which is the base of the project) and then we developed a technological prototype that proves everything we had in mind was possible to be developed.

1.2. First iteration

During the early weeks of October, the goals and objectives to be achieved were set, after a thorough analysis of the difficulties and risks related to a 3D approach. Despite a lot of work from <e-Adventure2D> was reusable (e.g. the parsers based on SAX technology), the <e-Adventure3D> project was conditioned by its dependence of an external 3D games engine. For that reason the aspirations for the first iteration were *reduced* (the term *reduced* is symbolic as the work and objectives scheduled during these two months and a half were enough to complete a whole-year Computer System Project (SI)) to the obtaining of a solid engine able to parse and execute an adventure storyboard, and an overall (but solid enough to support updates and further programming works) design. The next points highlight the features of the engine considered as deliverables at the end of the iteration:

- Ability to load rooms, objects and characters represented as models from the storyboard XML file.

- Handling of input events to allow the movement of the player.

- Basic interactions between the player and objects.

- Basic conversations with characters.

- Management of the *flag system*

- Effects triggering (change camera effect, triggering conversations, activate and deactivate flags, change the scene effect, consume object at the scene and generate objects at the inventory)

1.3. Second iteration

The main goal of the second iteration was to design and implement the *editor tool* for <e-Adventure3D>. We also pretended to extend the engine *effects system*.

We concluded the iteration with these goals achieved:

- Engine effects system extension: cut-scenes, sounds and books.

- Editor tool: We developed the editor that allows the user to create games as solid as they were in first iteration but in a graphical way.

1.4. Third iteration

For the third iteration were left the *reports generation* (assessment), *communication with a Learning Management System* (LMS) and improvements and extensions of the engine and the editor tool.

1.5. Final iteration

This final iteration goal is to smooth things over: finish with last improvements and complete the documentation.

On the next sections we describe the planning and schedule followed, structured in one, two or three weeks long micro-iterations, how both storyboard specification language and engine-editor were updated to address these goals, and finally some lines of discussion about the objectives accomplished at the end of the iteration and future work.

2. Work Process

This section contains a detailed description about the work process during the project. We are going to contrast it with what we planned at the beginning of the project. As we mentioned, the work process was divided in five iterations as we can see in the next diagram:

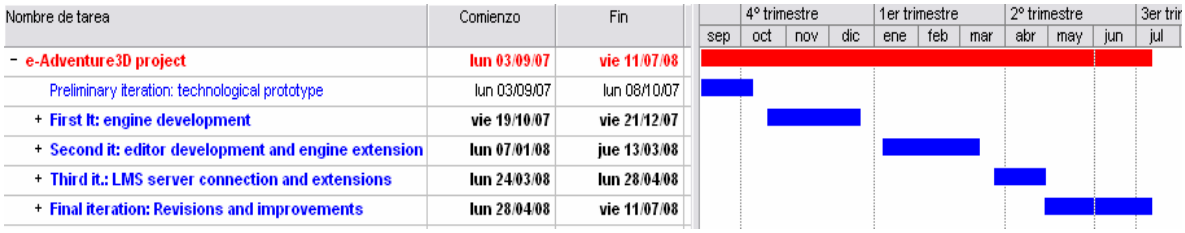


Figure 7. Main Gantt Diagram of the project

For each of the main iterations, we are going to explain the work process in a detailed way. At first we are going to show the Gantt diagram to remember the micro iterations that compound the iteration and then we are going to explain the work process that has been made in each of them and the problems we found during the process. We are not going to detail the concepts we have developed, because a detailed explanation of them belongs to section "Characteristics of <e-Adventure3D>".

2.1. First iteration: Engine development

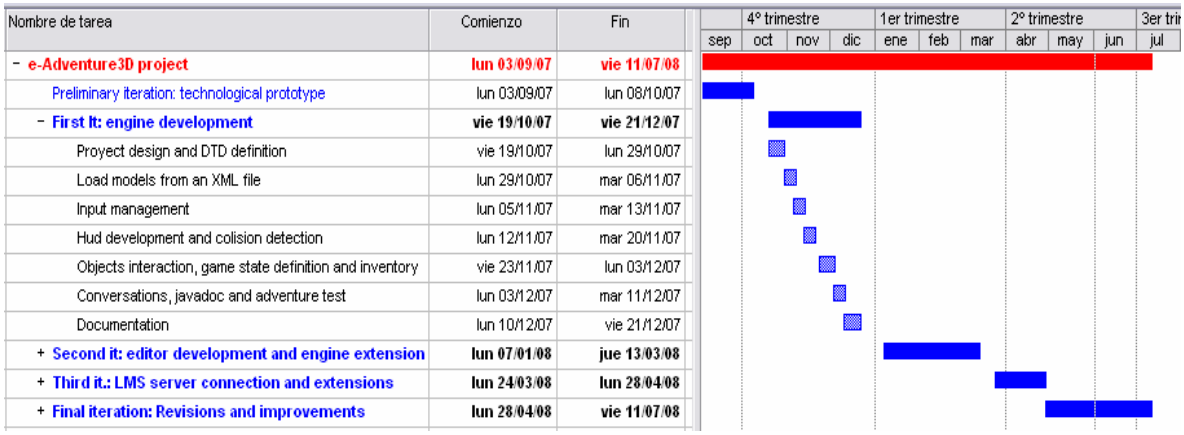


Figure 8. Gantt Diagram of the first iteration

The first iteration started on 23rd of October and ended on 27th of December. Although during this period we started the real work related to the <e-

Adventure3D>, as we have written before it was not the first contact with the project and its idiosyncrasy. Previously, we developed a small graphical adventure using the JMonkey (JME) so we took contact with the technologies we were considering to use. The goal of developing the prototype was to ensure that such technologies could cover all the project requirements. Then, back on 23rd of October we started the real design and implementation tasks.

The aim of the first iteration was to support the creation of 'basic' graphical adventures in a three dimensional space from a storyboard written in XML. That load of work was divided into seven different micro iterations:

Micro iteration 1: DTD definition and code structure design

Firstly, we updated the <e-Adventure2D> language so it could support the new requirements of the project; that is, we redefined the DTD.

Secondly, before starting the programming work, we designed a clear structure of the code to ensure that we were going to follow the pattern model-view-controller.

Micro iteration 2: Load models and scenes from an XML file + development of game cameras.

The aim of this micro iteration was to load 3d models from an XML file. The objective was achieved, and at the end of this period the engine was able to load 3D models of characters and items and we also were capable of generate scenes (open environments and closed rooms) from data parsed from an XML file which stores the storyboard of the game.

We also develop a first version of one important part of 3D games that we didn't consider at all in the original plan: cameras. At the end of this micro iteration cameras worked but we had to review some things in next micro iteration, such as collisions with walls in the closed rooms.

Micro iteration 3: The input system (first part)

One of the greatest changes regarding to <e-Adventure2D> was the management of the input system. In this project we leave the mouse to a side, because our games are controlled with a game pad or a keyboard.

We found some difficulties in our way like the poor management of the game pads offered by JME, or the big amount of differences between different game pad

models and brands. This last problem forced us to develop a *game pad configuration tool*, to provide an easy utility for the users to get their own game pads configured, as the JME did not do it automatically. The situation forced us to divide the work in order to end according to the plan, so while one of us was developing the game pad input system, the others continue with the next micro iteration. The problem was that the next micro iteration needed the input system to be done. We solved this using the keyboard, which is easier to program in JME games.

Micro iteration 4: HUD + collision detection + the input system (part 2)

Once the movement of the player was implemented with the keyboard, and models and environments became loadable, we started working on the interaction between them. We ensured the player could not walk through the objects by designing and implementing a *Collision Detector Module* and improve the implementation of the third person cameras to make sure that a third person camera does not pass through walls while the player is walking along a closed room. We also implemented the HUD, which shows the accessible elements and their information in each current moment of a game.

At the end of this micro iteration the input system was also finished and games could be played with the game pad as well as they were played with the keyboard.

Micro iteration 5: Inventory + objects interaction + effects + changes of scene + game states

This was one of the longest micro iterations because it covered many topics. At first, we designed and developed a very important feature, the *inventory*. Once it was done and working properly, we adapted *the flag system* from the original <e-Adventure2D> for the elements we have in the new version, such as cameras or transformations of the models. The work went by more easily than we expected (thanks to our great efforts and exhaustive work), so we decided to extend the aims of the iteration, so the *effects and changes of scene* (that objectives were thought for the second iteration) were implemented.

Then the idea of a *"game state"* arose as a necessity. Now, we consider different states of the game such as a playing state, run effects state, change scene state, etc., so that we can manage the different states where an adventure can be.

Micro iteration 6: Conversations

The goal of this micro iteration was to introduce the conversations in the

adventure. As the original <e-Adventure2D> allows, there are *graph conversations and tree conversations*. We added a new game state, the game state conversation to manage the game when a conversation between two characters is taking place.

Micro iteration 7: Documentation of the first iteration.

The goal was to document all the work done during first iteration. We completed the javadoc of the code and the first version of this report.

We also put to the test the project with an instance of a chemistry educational game (see the chapter "Caso de estudio"), and the results obtained were quite satisfactory.

2.2. Second iteration

Nombre de tarea	Comienzo	Fin
− e-Adventure3D project	lun 03/09/07	vie 11/07/08
Preliminary iteration: technological prototype	lun 03/09/07	lun 08/10/07
+ First It: engine development	vie 19/10/07	vie 21/12/07
− Second it: editor development and engine extension	lun 07/01/08	jue 13/03/08
Editor: design	lun 07/01/08	lun 14/01/08
Engine: unresolved tasks from first iteration	lun 07/01/08	lun 14/01/08
Editor:first steps with scenes edition	lun 14/01/08	mar 22/01/08
Engine: sounds in conversations	lun 14/01/08	mar 22/01/08
Editor: complete scenes edition	lun 21/01/08	mar 29/01/08
Engine: game menu (load and save)	lun 21/01/08	mar 29/01/08
Editor: characters and items edition	lun 28/01/08	mar 05/02/08
Engine: cutscenes	lun 28/01/08	mar 05/02/08
Editor: conversations edition	lun 04/02/08	mar 12/02/08
Engine: books	lun 04/02/08	mar 12/02/08
Editor: conditions and effects edition	lun 11/02/08	jue 28/02/08
Engine: improve effects system	lun 11/02/08	jue 28/02/08
Editor: documentation	lun 25/02/08	jue 13/03/08
Engine: documentation	lun 25/02/08	jue 13/03/08
+ Third it.: LMS server connection and extensions	lun 24/03/08	lun 28/04/08
+ Final iteration: Revisions and improvements	lun 28/04/08	vie 11/07/08

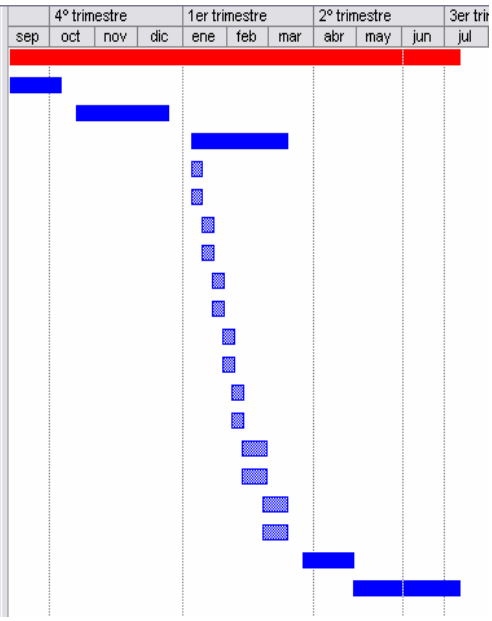


Figure 9. Gantt Diagram of the second iteration

One of the most arduous parts of the first iteration was to write the storyboard of the test game. We realised the necessity of an editor which allows the user to create the adventures in a visual way so he does not have to write anything because the XML file is created automatically.

As we can see in the Gantt diagram there were two work lines in this iteration. The first one was the implementation of the editor and its aim was to create an editor which allows the user to create games as complete as they were in the first iteration. The second work line was about some necessary extensions in the

engine; we believe in the importance of voiced conversations in 3D games and the utility of books, slides and videos in educational games.

As the Gantt diagram indicates, this second iteration was divided into seven double micro iterations. However, some differences between the plan and the work process can be noticed because we found some problems during the editor development at the beginning. Next we are going to detail each of the micro iterations:

Micro iteration 1: Editor: design + Engine: solving problems

The first step was to specify the code structure and the view design of the editor. We separated the structure of the XML file from the aspect of the editor in order to make the life easier to users. They will find a functional and efficient editor which its use is completely independent of the <e-Adventure3D> language.

Speaking about the engine, we fixed some small problems that we discovered while we were developing our first adventure.

Micro iteration 2: Editor: first steps with the view + Engine: improving conversations

Once the editor was designed, we started to develop it. Firstly we put in order the packages and the main classes of the code as we specified in the previous micro iteration applying as always the model-view-controller pattern. Secondly we started the development of the view of the scenes edition as we specified in the first micro iteration.

In parallel we started with some extensions of the original engine; specifically we improved the conversations we mainly allowed the possibility of playing sounds and launching effects in each line of a conversation. The big deal was to get to return to the conversation after the launch of an effect, so that we had to create a new state of the game: game state launching effects during a conversation. This extension wasn't planned at first but we find it very useful.

Micro iteration 3: Editor: some problems with JME + Engine: the game menu

The scenes are the most complicated part of the edition of adventures, so we needed another micro iteration in order to continue what we started in the previous one. Moreover, to complete the development of the scenes we need to

have characters and objects (in order to place them in the scenes), so we developed a first version of the edition of characters and items in the editor.

At this point, we realized that the development of the editor was more complicated than we had planned. Mainly we discovered these unexpected problems:

- Problems with the Z-order: JME components don't respect the z-order inside a Java Swing component so JME screens are always drawn above the other components. We were told by the JME developers that there is no solution for the moment to this problem so we need to take care of it trying not to place anything over a JME screen (for example the menus at the top of the editor's main window).

- Problems with layouts: JME is a new technology and its main purpose is not to be used inside Java components. So, when we used predefined layouts of the swing components (such as GridLayout, BorderLayout or even GridBagLayouts), we found that our JME screens were located without sense. There was just one solution to solve this problem: we had to create our own layouts.

In regard to the engine, we added a game menu. The menu allows users to go out of the game, change conversation options and load or save games. In order to do this we use another game state.

Micro iteration 4: Editor: Completing the view + Engine: Cut scenes

Once we knew the limitations of the JME while it is used inside swing components, it was easier to finish the implementation of the scenes, items and characters edition. Even so, we found some difficulties and we thought about some extensions that make us to spend more time developing this part.

In the engine, we created the other type of scenes named "cut scenes", that are videos and slides. Once again, we had some problems related with JME. This time the problem appears with videos, because the engine does not allow playing videos in games, so we had to use some external libraries.

Micro iteration 5: Editor: Check and adding conversations + Engine: Books

We use this micro iteration to ensure that everything was going fine. The items and characters that users create must be accessible in the scenes as well as the player. Moreover, the transformations users applied to characters, items and player should be represented in their instances in the scenes. In conclusion, one of the goals of this micro iteration was to merge the edition of the scenes with the edition of characters, items and player. The other goal was to add the conversations editor in order to create conversations in a graphical way. It was not a big deal because conversations are very similar as they were in <e-Adventure2D> so we only could reuse code.

This micro iteration was used in the engine to allow the use of books.

Micro iteration 6: Editor: Adding conditions and effects, save and load + Engine: Checking new extensions

This was one of the longest micro iterations because it has many different goals. Firstly, we make possible to add conditions and all kind of effects we supported in the engine where the <e-Adventure3D> language allows, for example conditions at references and exits in the scenes or effects after an action. Secondly we administered the control of ids. We had to make sure that our adventures are solid so, for example, if the user deletes an item (or changes its id) we have to delete automatically all its references (or change the ids) in the scenes and effects. After reaching the mentioned goals, we were prepared to save the information contented in our data classes into the XML file which represents the storyboard of the adventure using DOM. We also need to load the data we previously saved so we developed this function at this point.

In regard to the engine, we found some new bugs while we test the new extensions and we used this iteration to solve them.

Micro iteration 7: Editor: ZIP and documentation + Engine: documentation

This was also a significant micro iteration. At first it was supposed to last only one week but we realised that we had enough time and it was necessary to improve some things before starting to document the iteration.

The first goal was to save and load the data of an adventure from a ZIP file (with

the extension changed to '.ea3d'). This file contents all files that compound an adventure: the XML files with the different chapters, an XML file with a descriptor (name of the adventure, chapters, and some other information) and all the assets used (such as models, textures and icons).

After doing this, we were prepared to write the report of this iteration and the javadoc of the code.

2.3. Third iteration

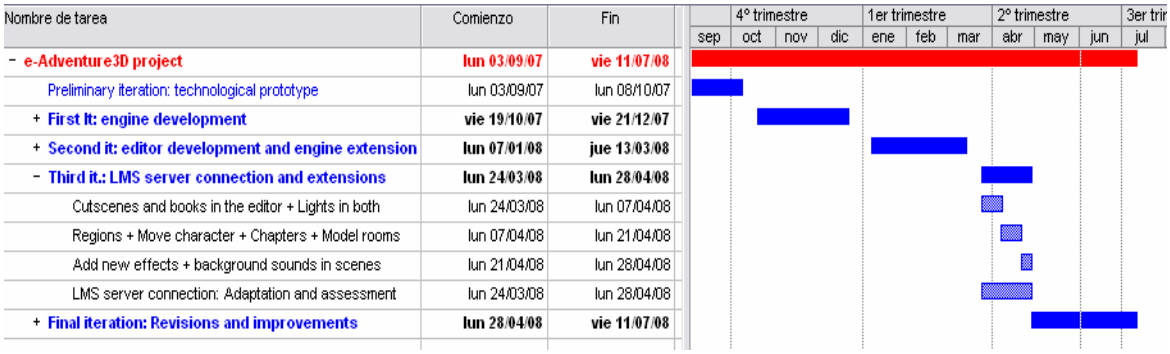


Figure 10. Gantt Diagram of the third iteration

As we did in the second iteration, we divided the work in two independent parts. One line of work was the inclusion in the engine of assessment and adaptation. The other work line was about new extensions in the engine and the editor. While we tested the chemistry game that we developed in the first iteration we thought about characteristics we would like to have in our games, so in this iteration we extended the engine and the editor with what we thought were the most useful ideas. We also extended the editor with the functionalities we added to the engine in the previous iteration. Next, we will detail the work done along the three micro iterations:

Micro iteration 1: Extensions of the editor based in previous engine extensions + Lights inclusion.

From the last iteration we had to allow the edition of slide scenes, video scenes and books in the editor. We also modify the conversations edition according to the extensions we had made to conversations in the engine.

We also allow editing the assessment and adaptation profiles in our adventures.

Until this moment, we had been using default lights in the scenes. We believe that lights can be very useful in educational games, not just because it allows users to see the models properly, but because lights can allow the game maker to

emphasize some parts of the scene so that users will be led during the game. In conclusion, at the end of this micro iteration we allowed the edition of lights with the editor tool and, the edition results, can be appreciated in the engine.

Micro iteration 2: Chapters + load from ZIP in the engine +Move character effect + Regions in the space + Model rooms scenes.

In spite of working hard, we had a lot of things to do in this micro iteration. This micro iteration was probably one of the most productive.

We wanted the adventures to have more than one chapter. This was considered when we create one adventure in the editor tool, but not in the engine. We made the necessary changes to allow this.

We have told that the adventures developed with the editor tool where saved in a ZIP file, so that we need to change the engine so the XML files the art assets were also loaded from a ZIP file.

Sometimes is important to locate characters where we want or simply move them to another part of the scene, so that we added a new effect called “move character/player effect”.

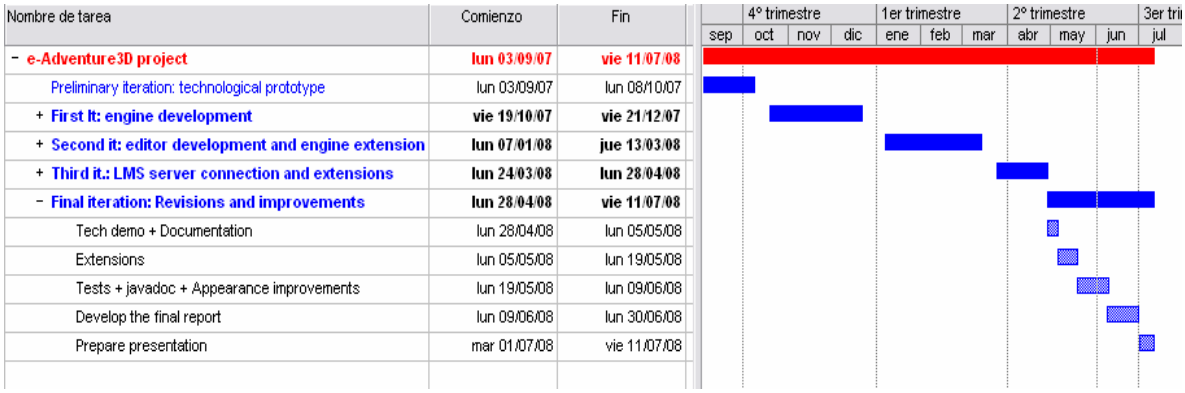
As we have written, while we were developing the chemistry game in the first iteration we thought about some ideas that can be very useful in our games. One of them was the concept of ‘region’ inside a scene, so that we extended the engine and the editor tool. We also decided to allow another kind of scene created from a 3D model.

Micro iteration 3: some new effects + background sounds in the scenes

Firstly we added the possibility of having a background sound during a scene. It was easy in the engine because JME allows the use of WAV and OGG sounds. However in the editor we had to use some extra libraries to allow this kind of sounds.

Secondly, we added a lot of new effects (in both engine and editor tool) related with the last extensions: end chapter, trigger book, trigger slide scene, trigger video scene, change animation, change light and play sound. We also modify the effect speak character to allow sounds in the sentence.

2.4. Final iteration



Nombre de tarea	Comienzo	Fin
− e-Adventure3D project	lun 03/09/07	vie 11/07/08
Preliminary iteration: technological prototype	lun 03/09/07	lun 08/10/07
+ First It: engine development	vie 19/10/07	vie 21/12/07
+ Second it: editor development and engine extension	lun 07/01/08	jue 13/03/08
+ Third it.: LMS server connection and extensions	lun 24/03/08	lun 28/04/08
− Final iteration: Revisions and improvements	lun 28/04/08	vie 11/07/08
Tech demo + Documentation	lun 28/04/08	lun 05/05/08
Extensions	lun 05/05/08	lun 19/05/08
Tests + javadoc + Appearance improvements	lun 19/05/08	lun 09/06/08
Develop the final report	lun 09/06/08	lun 30/06/08
Prepare presentation	mar 01/07/08	vie 11/07/08

Figure 11. Gantt Diagram of the final iteration

At this moment, our software was sufficiently complete to allow users to create their own educational 3D games with the editor tool and play them with the engine. The purpose of this iteration was to check that everything worked as it had been planned. We also wanted to develop some appearance improvements and, if we would have had time some extensions, too.

Micro iteration 1: Tech demo

This was the time to put into the test the editor tool and check that the games made with it can be played without incidents in the engine. The best way to do this and at the same time show the potential of the software was to create a tech demo. In this case, the tech demo was an educational game that teaches users what is <e-Adventure3D>, how can they use it and what can they do with it. We use everything we had developed in this game: object interaction, all different types of scenes, conditions and effects, etc.

Micro iteration 2: Solving bugs

While we were creating the demo tech with the editor and testing it in the engine, we discovered some small bugs we had to solve. So we had to debug the code instead of extend anything new as we had planned in the project planning. We used these two weeks to make sure everything works fine and do some optimizations (these optimizations are detailed in the section “Engine optimization process”).

Micro iteration 3: Appearance improvements + Final Report (first part)

This was the last micro iteration where we changed the code. We only made some appearance improvements. Mainly in the editor where we added icons in the

buttons and increase the size of the JME screens.

We wrote a first version of the final report to hand in it to our director teacher.

Micro iteration 4: Final Report (second part) + Test+ Java doc

We extended the report with some test of the software and looked over it so we can give in it to the secretary's office.

Micro iteration 5: Prepare presentation.

We will spend these ten days preparing the final presentation as we had planned.

3. Engine optimization process

Once the main development tasks of engine and editor were accomplished, we moved to the test, analysis and debugging phase. As in most of the cases, 3D video games are very complex pieces of software which performs an extensive consumption of the resources of the machine. Therefore, the development of this kind of applications must be carried out thoroughly, as a piece of code which is not properly optimized could lead to memory leaks, or to an excessive consumption of the processor or the memory.

In our case, we started by developing a simple tech demo, pursuing the goal of getting all the features of the engine tested. Then a performance test was carried out during the execution of the game, monitoring the status of both memory and processor. Next the composition of the game test is described from the point of view of its complexity, along with the results obtained.

3.1. Game Test.

Title: "Demo Técnica".

Number of chapters: 4.

Total size in Mb (including xml files and assets): 126 Mb.

Size per chapter (average in Mb): 46.02 Mb/chapter

Scenes per chapter (average): 1.75

Cutscenes per chapter (average): 3

3.2. Analysis of the results.

The test was executed in two different machines with similar results:

Machine A) Processor Intel® Core™ 2 Duo CPU T7500 2.2Ghz, 2046Mb RAM

Machine B) Processor Intel® Pentium IV Centrino 1.7 GHz, 1024 Mb RAM

The results obtained in both machines were similar, as expected. Hence the data provided in this section is what we obtained executing the engine in machine A.

At a first glance, the system apparently slowed down after a couple of minutes playing. That is, the movement of the player stopped to be smooth and continuous, becoming discontinuous, as if the player was moving in stumbles. A primary analysis of the memory revealed that when the engine was in the state PLAYING (the user has the control and can interact with the scene) the use of memory increased at a rate of approximately, 10Mb/s (even if the player was standing and not moving around). Obviously, such a high rate made the system to slow down considerably and, after a couple of minutes, to break due to an "Out Of Memory Error" reported by the JRE.

Moreover, it was noticed that the data loaded for each scene was stored in memory permanently, although those resources were released after a change of scene. Both issues made the game unplayable. When the game reached the third chapter the total memory of the process ascended up to 1400 Mb approximately, forcing the OS to increment the virtual memory, slowing down the performance of the system and, finally, breaking it due to a lack of memory. The next figure was taken through a performance and memory analysis tool (JProbe 7.0.3 evaluation version), and shows the evolution of the memory during the first three chapters of the test game.

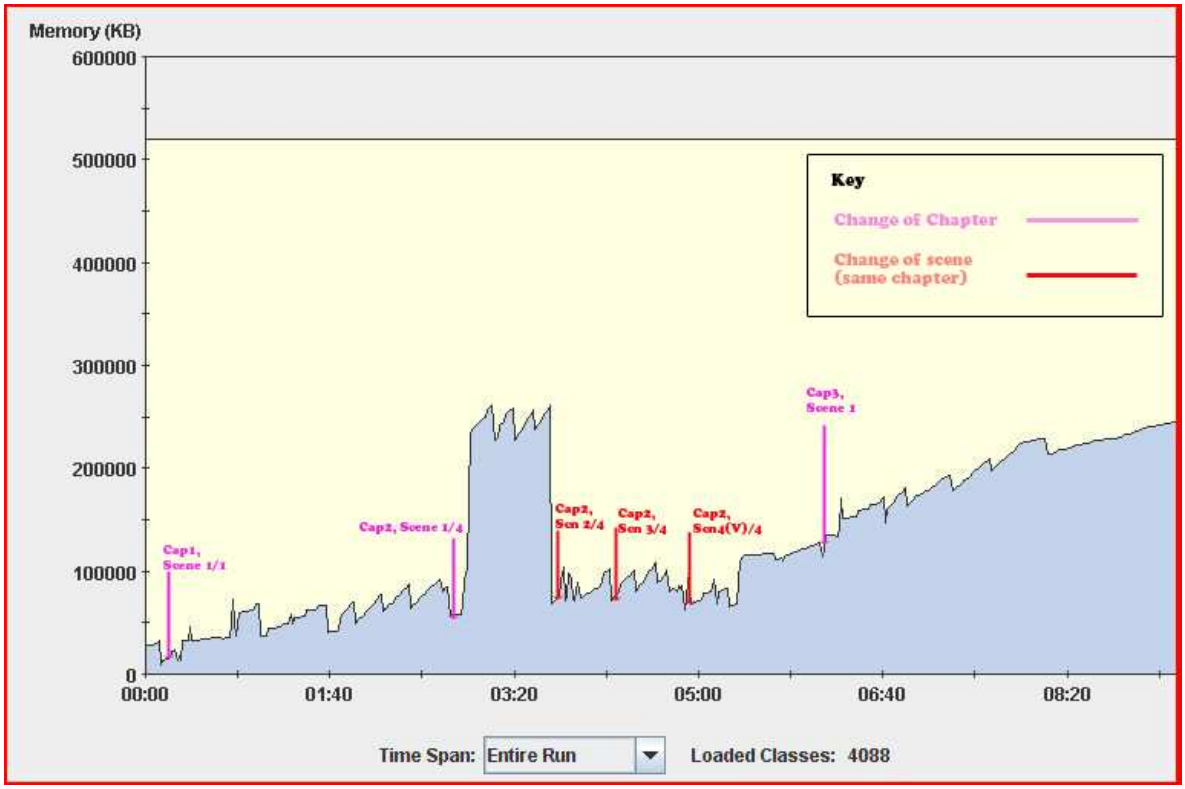


Figure 12. Memory status during the execution of the test game (without optimization).

As the graphic shows, the use of memory tended to increase in time. Especially during the first scene of the third chapter, which is the scene with more "intractability" (i.e. the scene which requires a longer time to be accomplished and therefore in which the PLAYING state was set longer).

Therefore, two main problems, which should be tackled in order to improve the use of the memory enough so the games could be played, were identified:

(1) *Stabilize the use of memory during the PLAYING state. That is, the net increment of used memory would remain constant over the execution of the same scene (average rate -> oMb).*

(2) *Clean the memory used in each scene after they are exited. That is, the net increment of used memory would remain constant in time, increasing when a scene is loaded but releasing all the data loaded when the scene is exited and before the next one is loaded.*

After a thorough analysis and effective debugging, the main causes of the problems (1) and (2) were found:

(1) *A wrong use of the swing components of the game (i.e. all the 2D panels used in the game: HUD, Inventory, Menu, Books panel, Conversations panel, Slides panel) was invoking the repaint() method of those panels too many times per frame. We could not find a proper explanation for this, as a priori an invocation of that method should not involve an extra consumption of memory, but repainting all the swing components in every frame produced a memory increment of 8.5 Mb/s approx.*

(2) *Although the scene-graph nodes were detached for every change of scene before the next one was loaded, somehow they persisted in memory during all the execution of the game. Therefore when the last chapter was reached the memory was collapsed with a lot of data that were no longer referenced. Something similar occurred with the loaded sounds (a heavy load for the engine, as all the sounds compounded the 80% of the total size of the game in Mb).*

3.3. Optimizations performed

Taking all these premises into account, we followed the next strategies to minimize the effects of both issues:

(1) *Minimize the calls to the repaint() method. Every swing component of the application should be forced to be repainted only when it really had changed:*

- The HUD panel only changes when the element selected in a certain moment varies and the available actions for this differ from the previous.
- The Menu panel only needs to be repainted during the MENU_STATE (not in the PLAYING_STATE), and only if the current sub-panel (conversation options, load / save menu, assessment report menu, etc.) changes.
- The Book panel only needs to be repainted in the BOOK_STATE, and just in case the current page displayed changes (e.g. the user moves from page 1 to page 2). Analogously the repaint() invocations if the Slides panel could be dramatically reduced.

- The Conversation panel only needs to be repainted in the CONVERSATION_STATE, and only if the options or dialogues of the conversation changes.

With all these reductions, the memory increment rate descended up to 0.2Kb / s. Finally we invoked the Garbage Collector each 20 seconds so now the net increment is virtually 0 Kb/s.

(2) *Although all the nodes in the scene were detached from the root node before loading a new scene, they were not removed from the memory by the garbage collector. After some tests the problem seemed to be in the tree nature of the detached Nodes. In this manner, the removed nodes were still being referenced by their children. We solved this by searching all along the tree and detaching each node from its parent. The leaf nodes (those which contain the data buffers) were cleaned, releasing all the data stored in them definitely.*

In addition, when a chapter is ended the Texture Manager is cleaned along with the sounds queue. After that an invocation of the Garbage Collector released effectively all the data loaded during the chapter.

A second monitoring of the memory during the execution of the same game during the same period of time proves that both goals were achieved. On the one hand, the memory remains stable in each scene (note that the time each scene is executed is delimited by the red and purple marks), even in the first scene of the third chapter. On the other, the memory used in each scene is released and not kept in memory during all the execution.

Consequently to this, the engine will run out of memory only if the data loaded in a scene is excessive (an issue that a priori we cannot foresee as it depends on the author of the game). After this optimization stage we can affirm that the engine is stable and usable.

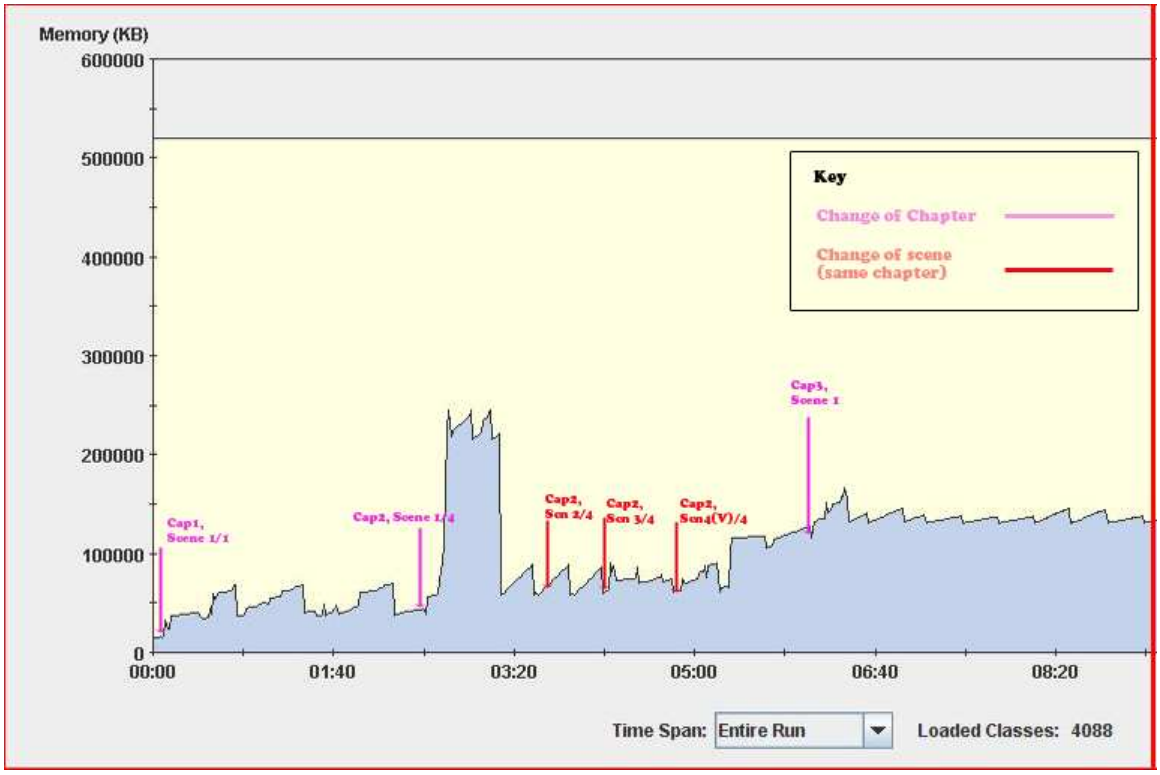


Figure 13. Memory status during the execution of the game test (after optimization).

4. Work distribution between project members

This project has been directed by **Baltasar Fernández Manjón** and **Pablo Moreno Ger**. The first one has coordinated the project in general terms, mainly reviewing the project evolution after each of the main iterations and taking care about the documentation. Pablo Moreno Ger has established the main objectives of the project planning. He also has supervised the development of the project each micro iteration through meetings with the development team.

Javier Torrente Vigil, Guillermo Cañizal Alzola and Ángel del Blanco Aguado have been the development team. In general terms, the engine has been implemented by the three of us and the editor tool and the chemistry educational game have been implemented by Guillermo and Javier. The DTD file that defines the chapters and the java doc commentaries in the code were written by the three of us (each one wrote the part he developed). Speaking about the report, Guillermo and Javier wrote everything in the report except for the editor tool manual, the case of study and the tests that were done by Ángel. Next we are going to specify the part of the work process developed by each member.

In the first iteration **Javier Torrente** was in charge of the input management system. He developed everything related with the movement of the player: the possibility of using a keyboard or any game pad to play games and the avoid collision system between the player and the other objects in a scene. He also developed everything related with the cameras. He was also in charge of the transformations that can be made to the 3D models. Speaking about loading 3D models he also created the predefined environment called closed room. The second iteration was about the editor where he developed firstly the same things he developed in the engine: transformations to 3D models and cameras management. Speaking about the editor, the scene edition that includes open environments and closed rooms were developed by him. Besides, he also included video scenes in the engine. He also learnt how to use the JME preview windows inside swing panels and developed the used of them. Speaking about items and characters he managed the load of them in the editor and the automatic saved of icons for the items. He was also in charge of the ZIP file and the assets control. In the third iteration he added books and the rest types of scenes to the editor tool: slide scenes and video scenes. He included assessment and adaptation in the editor tool. He added the use of the ZIP and, so that, the assets controller in the engine. He extended the editor tool and the engine with the other environment type, the

model room. He also added the regions to the scenes in both: the engine and the editor. The move player/character effects were developed by him in this iteration. In the last iteration, while Guillermo was developing the demo Tech, Javier was solving all the bugs found. He also took care of all the improvements in the view of the editor such as new icons, layouts, etc.

Guillermo Cañizal defined the environment called open environment at the first iteration. He also was in charge of the development of the HUD and the inventory. These two parts are related with the actions that can be applied to an item, so that he developed the load of items as well as the load of characters from the XML file. Actions also depend on conditions and effects so he added to the engine conditions and the initial version of the effects system with the effects related with this part. The possibility of changing from one scene to the next one was also developed by him. In the second iteration he was in charge of the editor with Javier. He developed many panels of the editor tool such us the one for the documentation, the animation panel, etc. He also took care about things he did in the engine such as the conditions, effects or the flags system. He also included the conversations edition in the editor tool. The ids control was also controlled by him. He wrote the DOM of the second iteration to save a chapter into an XML file. In the third iteration he improved the slide scenes in the engine with Javier's help. He also extended the engine and the editor tool in order to support chapters and a descriptor. As an extension for the engine and editor tool, he added the lights. He also added the background sound to the scenes and changed the edition of conversations to support sounds and effects. In the last iteration Guillermo was in charged of testing the editor tool. So that he developed the demo Tech.

In the first iteration **Ángel del Blanco** was in charge of the conversations and helped with the load of characters. He also developed a graphical interface to configure the game pad. In the second iteration he took care of the engine. He developed the use of books and the game menu. He included the slide scenes with an initial version. He also extends conversations in order to use sounds and effects in each line. In the third iteration he was in charge of assessment and adaptation in the engine.

Chapter XI

Tests

Two complete games (the chemistry game we described in the chapter “Caso de estudio”- Case Study - and the demo tech) were developed and played with the platform. One of the main objectives of developing these videogames was to make sure that every characteristics of <e-Adventure3D> worked properly. However, we also decided to do a few test more thoroughly of each specific characteristic. In order to standardize the tests we filled the same table for each of them:

Test	**Load #1**	
ID	**#LOAD1**	
Module (s)	Engine: GameStateLoadData	
Main goal	Test load game from saved data between chapters	
Objective (s)	Test load game from one chapter to another chapter	
	Test save game	
	Test the initial position when the game is loaded	
Description	Test Game: an adventure with 3 chapters. The game is saved in second chapter, in different scenes. The game will restart and try to load it. After the load, the game must be in the correct chapter.	
Parameter (s)	`Save file testLoad, testLoad2 and testLoad3`	
Succeeded	YES	After the load, is the game in the correct chapter?
	YES	Is the player in the position where was save?
	YES	Is the game correctly loaded in different scenes in the same chapter?
Failures (if founded)		
Suggested solutions		

Test	**Load #2**
ID	**#LOAD2**
Module (s)	Engine: GameStateLoadData
Main goal	Test load game from save data
Objective (s)	Test load game from one scene to another scene
	Test save game
	Test the initial position when the game is loaded
Description	Test Game: an adventure with 3 chapters. The game is saved in first chapter, in different scenes. The game will restart and try to load it. After the load, the game must be in the correct chapter.
Parameter	`Save file testLoad1,testLoad2 and testLoad3`

(s)		
Succeeded	YES	After the load, is the game in the correct scene?
	YES	Is the player in the position where was save?
Failures (if founded)		
Suggested solutions		

Test	**Load #3**	
ID	**#LOAD3**	
Module (s)	Engine: GameStateLoadData	
Main goal	Test the load parameters between scenes and chapters	
Objective (s)	Test the correct load of flags and conditions	
	Test the correct load of all models	
	Test the correct load of inventory data	
Description	Test game: an adventure with two chapter, and two scenes per chapter. In each scene there are three objects. One of these objects has one condition: it cannot be grabbed if the other two objects were not caught first. Also, when the game is loaded, must preserve the grab objects in the inventory.	

Parameter (s)	`Model of three objects:`
	`  - Plant.ms3D`
	`  - Globe.jme`
	`  - Book.ms3D`
	`Save file testLoad1 and testLoad2`	

Succeeded	YES	Is the game correctly loaded when grab objects?
	YES	Are all the grab objects in the inventory when the game is loaded?
	YES	Are the conditions and flags correctly loaded?
Failures (if founded)		
Suggested solutions		

Test	**Load #4**
ID	**#LOAD4**
Module (s)	Engine: GameStateLoadData
Main goal	Test the correct load of cameras and lights between scenes and chapters
Objective (s)	Test the cameras in a load from file game
	Test the lights in a load from file game

Description	Test game: an adventure with two chapters, and two scenes per chapter. In second chapter there are a change of lights when an object is grabbed. In the other scene of this second chapter, there is a change camera when an object is grabbed.	
Parameter (s)	- Camera1 (static camera): direction x="1.0" y="-0.3" z="1.0" position x="0.0" y="10.0" z="0.0" - Camera2 (third person camera): altitude="18.095745" angle="1.6707963" distance="35.751057 - Light1 (bulb light) position x="0.0" y="30.0" z="0.0" light-color alpha="0.95" blue="1.0" green="1.0" red="1.0" - Ligth2 (Lantern light) position x="0.0" y="15.0" z="0.0" direction x="0.0" y="-1.0" z="0.0" angle degrees="30.0" light-color alpha="1.0" blue="1.0" green="1.0" red="1.0"	
Succeeded	YES	Are the cameras correct when the game is load from file, between scenes and chapters? Are this true with both cameras?
	NO	Are the lights correct when the game is load from file, between scenes and chapters? Are this true with both lights?
Failures (if founded)	The lights there aren't the correct. The problem is that the lights aren't saved correctly.	
Suggested solutions	We save the lights from the class GameController through ChapterData. When this test is done again, it runs properly.	

Test	**Video Scene #1**	
ID	**#VDSCN1**	
Module (s)	Engine: GameStateVideo	
Main goal	Test the different type of video files that is supported	
Objective (s)	Test file with the extension ".AVI"	
	Test file with the extension ".MOV"	
	Test file with the extension ".MPG"	
Description	Test game: an adventure with one chapter, one interaction scene and three video scenes, one with each type of file extension. This type of scene was triggered grabbing one object.	
Parameter (s)	Test.avi, Test.mov and Test.mpg	
Succeeded	YES	Is played test.mov?
	YES	Is played test.mpg?

	NO	Is played test.avi?
Failures (if founded)	The file test.avi is not played because there are not the correct codecs installed in the system.	
Suggested solutions	Installing the correct codecs the problem was resolved.	

Test	**Slide Scene #1**	
ID	**#SLDSCN1**	
Module (s)	Engine: GameStateSlide	
Main goal	Test if the slide scene are correctly triggered	
Objective (s)	Test if the slide scene are correctly triggered	
	Test if the next scene after the slide scene is correctly load	
	Test if the next chapter after the slide scene is correctly load	
Description	Test game: an adventure with two chapter, one interaction scene, and two slide scenes. One slide scene takes to the interaction scene. The other slide scene takes to the other chapter. This slide scenes will be triggered when the player grab an object that appear in the interaction scene.	

Parameter (s)	`-Slide1`
	`  Next scene: principal scene`
	`-Slide2`
	`  Next scene: end chapter`	

Succeeded	YES	Are the slide scenes correctly triggered?
	YES	Is the next scene after slide1 the correct?
	YES	Is the next chapter after slide2 the correct?
Failures (if founded)		
Suggested solutions		

Test	**SlideScene #2**
ID	**#SLDSCN2**
Module (s)	Engine: GameStateSlide
Main goal	Test if the slide scene are correctly triggered
Objective (s)	Test if the slide scene is the first scene of the game
	Test if the slide scene active the correct flags (by means of effects)
Description	Test game: an adventure which begins with a slide scene. There is one chapter, with one interaction scene. There are an object and a character in this scene. When the player grabs the object, the slide scene is triggered, and after this, a conversation between the player and the character must

	begin, and generate a book in the player's inventory.	

Parameter (s)	`- Slide1`
	`  Next scene: interaction scene`
	`  Start: yes`
	`- Slide2`
	`  Next scene: interaction scene`
	`  Effects:` trigger-conversation idTarget="talkWithCharacter"
	generate-object idTarget="book"	

	YES	Is Slide1 the first scene in the game?
	YES	After Slide2, Is triggered the conversation "talkWithCharacter"?
	YES	After Slide2, Is generated the book in the player's inventory?
Failures (if founded)		
Suggested solutions		

Test	**Book#1**	
ID	**#BOOK1**	
Module (s)	Engine: TriggerBookEffect	
Main goal	Test the book	
Objective (s)	Test if the book is correctly triggered	
	Test if appear all the paragraphs along the pages of the book	
	Test if the images appear correctly	
Description	Test game: an adventure with one chapter and one scene. In this scene appears an object, and when is grabbed, a book is triggered.	
Parameter (s)	`Book:` `- Contains 8 title paragraph, 5 bullet paragraph, 9 text paragraph and an image`	
Succeeded	YES	Is the book correctly triggered?
	YES	Does the book contain all the paragraphs in correct order along the pages?
	YES	Does the book show the whole image?
Failures (if founded)		
Suggested solutions		

Test	**Open Environment Scene #1**
ID	**#OESCN1**
Module (s)	Engine:
Main goal	Test the elements in open environment scenes

Objective (s)	Test if all elements are in their positions	
	Test if light appear	
	Test if background sound is played	
Description	Test Game: an adventure with one chapter and one open environment scene. A background sound is attached. It scene has four elements of each type (object and character). Also there is a sun light.	

Parameter (s)	`-Sound: backgroundSound.ogg`
	`-Sun light`
	`direction x="-40.0" y="100.0" z="-40.0"`
	`light-color alpha="1.0" blue="1.0" green="1.0" red="1.0"`	

Succeeded	YES	Is each object in their position?
	YES	Is the sound played?
	YES	Does the sun light paper?
Failures (if founded)		
Suggested solutions		

Test	**Open Environment Scene #2**	
ID	**#OESCN2**	
Module (s)	Engine	
Main goal	Test the terrain in open environment	
Objective (s)	Test different types of terrains	
	Test if the player walks properly in each terrain	
Description	Test game: an adventure with one chapter and five scenes. In each scene appears an object. When is grabbed, the scene changes.	

Parameter (s)	`-Plain terrain`
	`-Smooth terrain`
	`-Mountainous terrain`
	`-Water terrain`
	`-Rough terrain`	

Succeeded	YES	Are all kinds of terrains correctly loaded?
	YES	Does the player walk correctly in each terrain?
Failures (if founded)		
Suggested solutions		

Test	**Open Environmentment Scene #3**	
ID	**#OESCN3**	
Module (s)	Engine	
Main goal	Test the skybox and textures in open environment	
Objective (s)	Test other skybox different from the editor's default	
	Test other textures in open environment	
Description		

Parameter (s)	`-Skybox1: citySky.jpg`
	`-Skybox2: mountainSky.jpg`
	`-Skybox3: nigthSky.jpg`
	`Each skybox has six files.`
	`-Textures: gras.png, sand.png and flowers.png`	

Succeeded	YES	Do citySky.jpg, mountainSky.jpg and nigthSky.jpg look like in the editor?
	YES	Do grass.png, sand.png and flowers.png?
Failures (if founded)		
Suggested solutions		

Test	**Room Scene #1**
ID	**#RMSCN1**
Module (s)	Engine
Main goal	Test the elements in room scenes
Objective (s)	Test if all elements are in their positions
	Test if lights appear
	Test if background sound is played
Description	Test Game: an adventure with one chapter and one room scene. A background sound is attached. It scene has four elements of each type (object and character). Also there are a bulb light and a lantern light.

Parameter (s)	`-Lantern light`
	`position x="0.0" y="58.0" z="-93.0"`
	`direction x="0.0" y="-1.0" z="0.0"`
	`angle degrees="45.0"`
	`light-color alpha="1.0" blue="1.0" green="1.0" red="1.0"`
	`-Bulb light`
	`position x="0.0" y="30.0" z="0.0"`

	light-color alpha="0.95" blue="1.0" green="1.0" red="1.0" -Sound: backgroundSound.ogg	
Succeeded	YES	Is each object in their position?
	YES	Is the sound played?
	YES	Do the both lights paper?
Failures (if founded)		
Suggested solutions		

Test	**Room Scene #2**	
ID	**#RMSCN2**	
Module (s)	Engine	
Main goal	Test to load a model room	
Objective (s)	Test if the model room are correctly load in a room scene	
	Test if the transformations in this model are the correct.	
Description	Test game: an adventure with one chapter and one room scene. That scene has a model room.	
Parameter (s)	-Model room: -Model modelRoom uri="assets/model/COLI_L" x="297" y="103" z="5" -Transformations objectRotation rotation-axis-x angle="-90.0" rotation-axis-y angle="0.0" rotation-axis-z angle="0.0" objectScale scale-axis-x scale="1.975897" scale-axis-y scale="1.975897" scale-axis-z scale="1.975897"	
Succeeded	YES	Is the model correctly loaded?
	YES	Are the transformations like in the editor?
Failures (if founded)		
Suggested solutions		

Test	**Third Person Camera #1**	
ID	**# CAM3RD1**	
Module (s)	Engine : CameraController	
Main goal	Test third person camera parameters	
Objective (s)	Test third person camera auto-recolocation	
	Test third person camera auto-restoration	
	Test third person camera parameters load from xml	
Description	Test Game: an adventure with one chapter, one room scene, and the player. The player is moved next to the four walls of the room, checking that camera never goes out of the room.	

Parameter (s)	`- Player: ninja (assets/model/ninja.ms3d)`
	characterDirection x="-0.0" y="0.0" z="-1.0"
	characterUpVector x="0.0" y="1.0" z="0.0"
	`- Camera:`
	- Distance: `8.962019`
	- Altitude:`12.793844`
	- Angle(rads): `1.8357964`	

Succeeded	YES	Camera auto-recolocates to fit in scene when near to all walls?
	YES	Camera auto-restores its position when goes far from a wall?
	YES	Camera is loaded exactly as it was displayed on the editor?
Failures (if founded)		
Suggested solutions		

Test	**Third Person Camera #2**
ID	**# CAM3RD2**
Module (s)	Engine : CameraController
Main goal	Test third person camera parameters for a scaled player
Objective (s)	Test third person camera auto-recolocation
	Test third person camera auto-restoration
	Test third person camera parameters load from xml
Description	Test Game: an adventure with one chapter, one room scene, and the player. The player is moved next to the four walls of the room, checking that camera never goes out of the room.

Parameter (s)	`- Player: ninja (assets/model/ninja.ms3d)`
	characterDirection x="-0.0" y="0.0" z="-1.0"
	characterUpVector x="0.0" y="1.0" z="0.0"
	
	`- Camera:`
	`ANGLE=1.8857963`

	ALTITUDE=28.232277 DIST=14.9456215	
Succeeded	YES	Camera auto-recolocates to fit in scene when near to all walls?
	YES	Camera auto-restores its position when goes far from a wall?
	YES	Camera is loaded exactly as it was displayed on the editor?
Failures (if founded)		
Suggested solutions		

Test	**Third Person Camera #3**	
ID	**# CAM3RD3**	
Module (s)	Engine : CameraController	
Main goal	Test third person camera auto-recolocation for long displaced cameras: distance (player, camera) is high	
Objective (s)	Test third person camera auto-recolocation	
	Test third person camera auto-restoration	
	Test third person camera parameters load from xml	
Description	Test Game: an adventure with one chapter, one room scene, and the player. The player is moved next to the four walls of the room, checking that camera never goes out of the room.	
Parameter (s)	- Player: ninja (assets/model/ninja.ms3d) characterDirection x="-0.0" y="0.0" z="-1.0" characterUpVector x="0.0" y="1.0" z="0.0" - Camera: ANGLE=1.5707964 ALTITUDE=10.0 DIST=46.0	
Succeeded	YES	Camera auto-recolocates to fit in scene when near to all walls?
	YES	Camera auto-restores its position when goes far from a wall?
	YES	Camera is loaded exactly as it was displayed on the editor?
Failures (if founded)		
Suggested solutions		

Test	**Third Person Camera #4**	
ID	**# CAM3RD4**	
Module (s)	Engine : CameraController	
Main goal	Test third person camera auto-recolocation for large player shrunkened)	
Objective (s)	Test third person camera auto-recolocation	
	Test third person camera auto-restoration	
	Test third person camera parameters load from xml	
Description	Test Game: an adventure with one chapter, one room scene, and the player. The player is moved next to the four walls of the room, checking that camera never goes out of the room.	

Parameter (s)	`- Player: dwarf (assets/model/oldman/dwarf.ms3d)`
	characterDirection x="-0.0" y="0.0" z="-1.0"
	characterUpVector x="0.0" y="1.0" z="0.0"
	
	`- Camera:`
	`ANGLE=1.6882963`
	`ALTITUDE=7.1033506`
	`DIST=8.758753`	

Succeeded	YES	Camera auto-recolocates to fit in scene when near to all walls?
	YES	Camera auto-restores its position when goes far from a wall?
	YES	Camera is loaded exactly as it was displayed on the editor?
Failures (if founded)		
Suggested solutions		

Test	**Third Person Camera #5**
ID	**# CAM3RD5**
Module (s)	Engine : CameraController
Main goal	Test third person camera auto-recolocation for model with direction !=(0,0,-1)
Objective (s)	Test third person camera auto-recolocation
	Test third person camera auto-restoration
	Test third person camera parameters load from xml
Description	Test Game: an adventure with one chapter, one room scene, and the player. The player is moved next to the four walls of the room, checking that camera never goes out of the room.
Parameter (s)	`- Player: dwarf (assets/model/zombie/zombie02.ms3d)` characterDirection x="-0.0" y="0.0" z="-1.0" characterUpVector x="0.0" y="1.0" z="0.0"

	- Camera: ANGLE:1.5707964 DIST:13.000001 HEIGHT:19.0	
Succeeded	YES	Camera auto-recolocates to fit in scene when near to all walls?
	YES	Camera auto-restores its position when goes far from a wall?
	YES	Camera is loaded exactly as it was displayed on the editor?
Failures (if founded)		
Suggested solutions		

Test	**Object #1**	
ID	**#OBJ1**	
Module (s)	Engine	
Main goal	Test objects	
Objective (s)	Test transformation in objects	
	Test other textures in objects	
	Test interaction with objects	
Description	Test game: an adventure with one chapter and one room scene. In this scene appears the same object with different transformations and textures. Each object has different effects associated to the actions.	
Parameter (s)	-Model: a4.ms3d -Transformations -Scale scale-axis-x scale="22.648933" scale-axis-y scale="22.648933" scale-axis-z scale="22.648933" -Rotation rotation-axis-x angle="-85.943695" rotation-axis-y angle="15.0" rotation-axis-z angle="13.08752" -Different textures: greyA4.jpg, tunning.jpg, race.jpg -Different associated effects (in different objects) to different actions: trigger-conversation idTarget="goodCar" activate flag="carOpen" generate-object idTarget="keys" trigger-book idTarget="carBook"	
Succeeded	YES	Do the transformations the same effect than the editor?
	YES	Do the textures appear with their associated object?
	YES	Do the effects run properly?

Failures (if founded)	
Suggested solutions	

Test	**Character #1**	
ID	**#CHAR1**	
Module (s)	Engine	
Main goal	Test characters and effects in conversations	
Objective (s)	Test transformation in characters	
	Test other textures in characters	
	Test interaction with characters	
Description	Test game: an adventure with one chapter and one room scene. In this scene appears the same character with different transformations and textures. Each character has different effects associated to conversations. Also, these conversations may have conditions.	

Parameter (s)	`-Model: a4.ms3d`
	`-Transformations`
	`-Scale`
	scale-axis-x scale="32.0"
	scale-axis-y scale="32.0"
	scale-axis-z scale="35.895021"
	`-Rotation`
	rotation-axis-x angle="40,7652"
	rotation-axis-y angle="-17.0982"
	rotation-axis-z angle="-6.0909"
	`-Different textures: viking.jpg, streetWear.jpg, soldier.jpg`
	
	`-Different associated conversations (and its conditions):`
	conversation-ref idTarget="conversacionChica"
	condition
	active flag="talk2firstCharacter"
	
	`-Different associated effects (in different characters) to different conversations:`
	trigger-slidescene idTarget="typeOfVikings"
	activate flag=" talk2firstCharacter "
	consume-object idTarget="book"	

Succeeded	YES	Do the transformations the same effect than the editor?
	YES	Do the textures appear with their associated characters?
	YES	Do the conversations and its associated effects take place when the conditions allow them?
Failures (if founded)		
Suggested solutions		

Test	**Regions #1**	
ID	**#REG1**	
Module (s)	Engine	
Main goal	Test effects in regions	
Objective (s)	Test transformations done in the editor in regions	
	Test effects and post effects in regions	
	Test conditions in regions	
Description	Test game: an adventure with one chapter and one room scene. In this scene is small, and here are big region. The regions cannot be seen, and take a tree as reference of the end of the region. When the player came into the region, effects must be triggered, and when he came out the region, post-effects must be triggered. The regions have conditions. The player must grab a book that appears in the scene. Then, if the player crosses the region, the effects will be triggered.	

Parameter (s)	`-Transformations (of the region)`
	`-Scale`
	scale-axis-x scale="55.9983"
	scale-axis-y scale="87.09723"
	scale-axis-z scale="100.0293"
	`-Rotation`
	rotation-axis-x angle="-73.038"
	rotation-axis-y angle="-23.9404"
	rotation-axis-z angle="-15.01934"
	
	`-Conditions (this flag are activated grabbing the book):`
	active flag="bookGrabbed"
	
	`-Different associated effects:`
	trigger-conversation idTarget="regionOk"
	generate-object idTarget="keys"
	trigger-book idTarget="Book"
	
	`-Different associated post-effects:`
	speak-player "regionOk"
	play-sound uri="sounds/sound.ogg"
	trigger-conversation idTarget="regionOk2"	

Succeeded	YES	Do the transformations the same effect than the editor?
	YES	Do conditions take effect?
	YES	Do the effects and post-effects work properly?
Failures (if founded)		
Suggested solutions		

Test	**Light #1**	
ID	**#LIGHT1**	
Module (s)	Engine	
Main goal	Test lantern light and bulb light	
Objective (s)	Test lantern light in a scene without lights (only ambient light, that is obligatory)	
	Test lantern light in a scene bulb lights	
	Test effect “Switch on/of light”	
Description	Test game: an adventure with one chapter and three room scenes. In the first scene, there are only a lantern light pointing a book over the table (both are objects), the ambient light is set dark. In the second scene, is the same but with a bulb light. In the third scene, there is a lantern in the same position, but also there is a bulb light. In this scene are 2 objects that symbolize switches. One has linked a “Change light” for bulb that switches on/off that light. The other has the same effect to the lantern light.	

Parameter (s)	`-Ambient light (obligatory in each scene):`
	`light-color alpha="0.0" blue="0.0" green="0.0" red="0.0"`
	
	`-Lantern light`
	`position x="0.0" y="30.0" z="0.0"`
	`direction x="0.0" y="-1.0" z="0.0"`
	`angle degrees="35.0"`
	`light-color alpha="1.0" blue="1.0" green="1.0" red="1.0"`
	
	`-Effect associated to each switch`
	`change-light idTarget="bulbId"`
	`change-light idTarget="lanternId"`	

Succeeded	YES	Does the lantern light work properly?
	YES	Do the two types of light work properly together? Are the lights in the same position than in the editor?
	YES	Do the effect the correct work?
Failures (if founded)		
Suggested solutions		

Capítulo XII

Caso de Estudio

1. Sobre el caso de estudio

Al terminar la primera iteración, desarrollamos una aventura gráfica escribiendo el guión 'a mano' en XML. Por una parte, el objetivo del juego era hacer que los usuarios asimilaran unos conceptos básicos de química y, por otra parte, tratar de utilizar en dicho juego todas las funcionalidades soportadas por el motor en ese momento. Tras las sucesivas ampliaciones en el motor a lo largo de las dos iteraciones siguientes (sobretodo en lo que a crear juegos educativos se refiere) surgió la idea de retomar este juego. Para ello, se rehizo en su totalidad el juego utilizando el editor y aprovechando las nuevas características educativas de la plataforma.

Creímos conveniente añadir este apartado en la memoria dado que es una confirmación empírica de las capacidades del proyecto. En primer lugar vamos a recoger información detallada sobre el videojuego desarrollado con la plataforma y, en segundo lugar, vamos a mostrar los resultados obtenidos del proceso de desarrollo y a analizarlos.

2. Guión del juego

2.1 Resumen del juego

2.1.1 Propósito educativo

El propósito de este juego es que el alumno que lo utilice adquiera nociones básicas sobre química. Se presentan conceptos básicos al inicio, tales como qué es la química, qué son los átomos, elementos, mezclas y reacciones químicas. También propiedades de algunos elementos y compuestos, como por ejemplo el hidrógeno, metano, ácido clorhídrico, hidróxido de sodio, etc, y qué resultados se producen al ser mezclados unos con otros. El juego nos permite navegar por tres escenarios en los que, con diversos medios (fundamentalmente conversaciones, libros y diapositivas), se pretende transmitir dichos conceptos.

A la vez que se desarrolla la aventura podemos evaluar la actividad del alumno en función de las acciones que realice en el mismo por medio de un perfil de evaluación automática ('assessment'). De esta manera se reflejarán tanto los pasos

correctos que tome el alumno, como incorrectos.

2.1.2 Descripción general

Ronin es un empleado de una importante empresa de química. Allí trabaja con Harriet, su novia. Un día se encuentra un poco mareado y se desmaya. Cuando despierta se encuentra en un cuarto en el que nunca había estado y solamente piensa en encontrar a su novia y salir de allí. En el extraño cuarto hay un profesor, el cual le da algunas pistas de cómo encontrar a su amada.

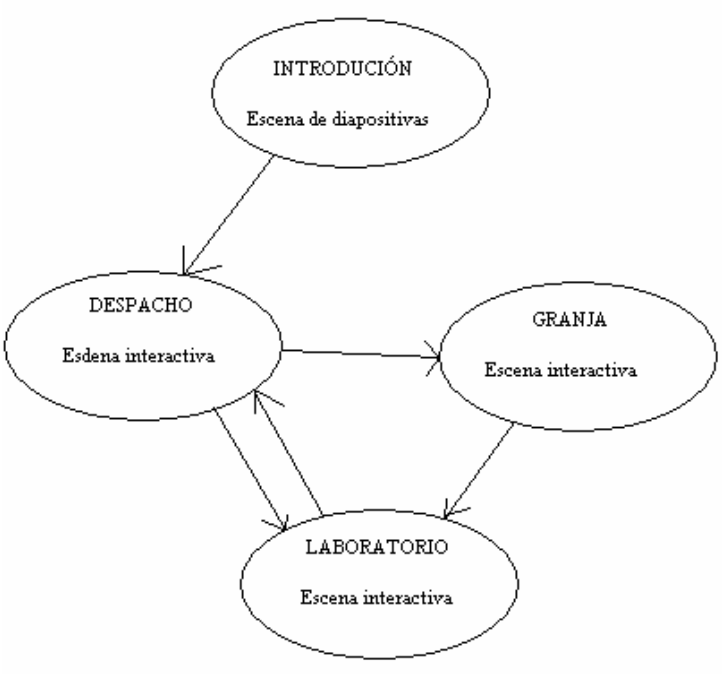
Ronin se da cuenta que, para salir de este embrollo tendrá que aprender nociones básicas de química, ya que el profesor le hizo ver que el Dr. Pipet (un reputado científico de la empresa que perdió la cabeza probando en sí mismo peligrosos experimentos) está detrás de la desaparición de Harriet. El profesor le dará la información teórica que necesita para enfrentarse a esta aventura. Lo primero que deberá hacer es coger varios objetos del cuarto para poder salir de el.

En la siguiente escena, Ronin se encuentra a las afueras de una granja, la cual tiene la puerta cerrada. Deberá interactuar con los animales que allí se encuentran para conseguir metano y almacenarlo en un globo que recogió en la escena anterior. Podrá abrir la puerta bloqueada si usa un mechero (que le regala el profesor) con el globo lleno de gas metano sobre la cerradura de la puerta.

En la última escena nos encontramos el laboratorio del Dr. Pipet. Este nos confirma lo peor, nos cuenta que Harriet se prestó voluntaria a probar uno de sus experimentos y quedó transformada en una horrenda de bestia. El doctor nos irá pidiendo que le traigamos distintos elementos químicos (que deberemos fabricar) y su cuaderno de notas para preparar el antídoto que devuelva a Harriet a su estado original. Por último, justo antes de que prepare el antídoto para liberar a Harriet, nos hará un pequeño examen para ver si hemos afianzado los conocimientos que se pretendía.

2.2 Escenarios del juego

A continuación mostramos el mapa de escenas de la aventura:



Despacho del profesor

Descripción:

El juego comienza en el despacho del profesor (tras una escena de corte de presentación), en el cual se encuentra Ronin después de su repentino desvanecimiento. Deberá hablar con el profesor para empezar a comprender como puede encontrar a Harriet. Este le preguntará sobre las nociones que el jugador tiene de química y, en función del grado de conocimiento previo a la aventura: si el alumno tiene unos conocimientos muy bajos nos mostrará una escena de corte con nociones básicas y le entregará un libro; si su conocimiento es de grado medio, le será entregado el libro solamente; si por el contrario se ve que tiene unos conocimientos altos de química no se le proporcionará ninguna información adicional. Si el alumno observa la mesa que hay al lado del profesor, y examina el globo que aquí encuentra, el profesor contará que en globos de este tipo se pueden almacenar gases, proporcionará información útil sobre el metano y le entregará un mechero. Deberá coger el globo y una planta que hay en el cuarto para poder salir por una de las puertas del cuarto (esta es una de las maneras de conducir la aventura a través de los 'flags').

Además el jugador deberá volver al despacho después de estar con el Dr. Pipet para recuperar un cuaderno que no encuentra. Para ello el jugador preguntará al profesor sobre el SO_3 y este contará como hacer ácido sulfúrico. Cuando el jugador

lo haya conseguido, lo echaremos en el cofre para que lo descomponga y se pueda coger el cuaderno.

Objetos:

- Globo: se encuentra en la mesa al lado del profesor. Debe ser recogido para poder salir de esta primera escena.
- Planta: en una esquina del cuarto se encuentra una planta. El jugador debe cogerla, ya que, como el globo, le será de utilidad en la siguiente escena, y es requerida para poder salir del cuarto del profesor.
- Cofre: este cofre se encuentra cerrado. Posteriormente se deberá utilizar uno de los compuestos que generemos para abrirlo y descubrir el cuaderno que hay en su interior.
- Cuaderno: el jugador tiene que entregárselo al Dr. Pipet, y tras ello le hará un examen para evaluar sus conocimientos. Si pasa el examen, podrá reencontrase con Harriet y finalizará la aventura.

Personajes:

- Profesor: hay dos conversaciones con el profesor, en la primera ("Introduction") da las nociones básicas sobre química, y en la segunda ("examineBalloon1") dará información sobre gases, mas en concreto sobre el metano, y dará un mechero que aparecerá en nuestro inventario.

Salidas:

- Salida a la granja: se encuentra en la pared a la izquierda del profesor, y nos llevará a la granja.
- Salida al laboratorio del Dr. Pipet: nos permitirá ir del laboratorio al despacho para que podamos acceder rápido al cofre y volver sin pasar de nuevo por la granja.

Detalles de implementación:

- Salidas condicionadas**:** Para poder salir hacia la granja es necesario que el jugador tenga en el inventario la planta y el globo. Para poder salir al laboratorio de Dr. Pipet solo estará disponible una vez que haya pasado

de la granja al laboratorio del Dr. Pipet.

- Implementación del globo**:** El globo que aparece en la mesa está implementado como un objeto con la acción "Examinar" asociada. Esta acción dispara una conversación con el profesor, en la nos da información sobre el metano y como explosionarlo. Además hará entrega de un mechero. Esto se consigue lanzando un efecto de generar objeto, el cual introduce el mechero en nuestro inventario. Una vez hemos pasado esta conversación, podemos volver a "Examinar" el globo y cogerlo. Ahora al examinarlo nos da una descripción sobre el globo, ya que el profesor ya nos entregó el mechero. Esto lo conseguimos mediante condiciones asociadas al objeto con el flag "examineBalloon" el cual se activará desde la primera conversación asociada al globo.
- Parte opcional**:** Podríamos haber asociado sonidos a cada línea de conversación, o haber metido cualquier sonido de fondo a la escena.

Granja

Descripción:

Nada mas entrar en esta escena, el jugador se encuentra una granja al fondo de la escena, la cual tiene la puerta cerrada. Estamos en una granja muy peculiar y podemos encontrar varios animales: un caballo, una cebra, un pterano (dinosaurio volador). También hay un cubo. Deberá observar el cubo para encontrar en su interior un libro que tiene una ampliación a los conocimientos sobre metano y explosiones que nos dio el profesor en la escena anterior. Después deberemos coger el cubo. Podemos intentar hablar con los animales, pero las contestaciones que nos van a dar, lógicamente, no tienen ningún sentido… Tendremos que ponerle el globo a la cebra y darle de comer, para que produzca metano. Con este globo podemos reventar la puerta de la granja utilizando para ello el mechero para prender fuego al metano. De esta forma entraremos dentro de la granja donde se encuentra el laboratorio del Dr. Pipet.

Objetos:

- Granja: el jugador tiene que abrir la puerta por medio del globo de metano y el mechero, lo que le dará acceso al laboratorio del Dr. Pipet.

- Cubo: si lo examinamos obtendremos un libro con información sobre metano y explosiones. Además deberá recogerlo ya que será utilizado en la siguiente escena para recoger agua.
- CH4: es decir, metano. Se generará cuando el personaje principal dé de comer a la cebra con la planta recogida en el despacho del profesor, y servirá para abrir la puerta de la granja y cambiar de escena.

Personajes:

- Caballo: este caballo se encuentra pastando tranquilamente por el césped que rodea la granja. Se puede intentar hablar con el, pero obviamente, no nos dará ninguna información relevante para el juego.
- Cebra: también se encuentra en el pasto que rodea la granja, pero no le gusta tanto la hierba que aquí se encuentra, por lo que cuando le damos la planta que encontramos la comerá rápidamente, y de su rápida digestión podremos obtener el metano que necesitamos para salir de esta escena.
- Pterano: es simplemente un reflejo de los experimentos que se llevan a cabo en la granja, no podemos interactuar de ninguna manera con él.

Salidas:

- Puerta de la granja: la única manera de salir de esta escena es por medio de la puerta de la granja, y por aquí nos dirigimos al laboratorio del Dr. Pipet.

Detalles de implementación:

- Salidas condicionadas: Para poder salir por la puerta de la granja, se necesita abrirla aplicando fuego con el mechero al globo lleno de metano, es decir, esto activará el flag necesario para entrar en la granja. Para conseguirlo, el jugador tiene que utilizar la acción “Usar con” entre el objeto CH4 que debe tener en el inventario y la granja. Después tiene que utilizar la misma acción entre el mechero y la granja, lo cual activará el flag “doorExploted”, que es el que condiciona la salida. Cabe destacar que también están condicionadas las acciones, por lo que realizarlas de manera inversa no da el mismo resultado.

- Implementación del cubo: Cuando el personaje principal se acerca al cubo solamente puede "Examinarlo". Al examinarlo, aparece un pequeño comentario del jugador indicando que ve un libro al fondo del cubo, y que lo va a leer. A continuación aparece un libro, con información que será útil para el examen final. Esto lo hemos conseguido mediante efectos asociados a dicha acción, de igual manera que en la escena anterior con el globo. Una vez examinado el cubo, se permitirá cogerlo. Esto lo hemos conseguido poniendo condiciones asociadas a las acciones.

Laboratorio del Dr. Pipet

Descripción:

Al entrar por la puerta de la granja el jugador da con el laboratorio del Dr. Pipet. Según entra, empieza una conversación con el doctor, ya que se extraña al ver un desconocido en su laboratorio. Ronin le cuenta que esta buscando a Harriet, y el Dr. Pipet le comenta que se prestó voluntaria para que probaran un experimento con ella, y que desafortunadamente salio mal, por lo que ahora esta convertida en bestia. Para volverla a su estado original, el Dr. Pipet le encomienda a Ronin que busque su cuaderno, ya que no sabe que ha hecho con él y ahí tiene apuntado la proporción de los elementos que forman el antídoto a su experimento. Además, para prepararlo, tiene que conseguir traerle sal y amoniaco. Ronin puede preguntar al doctor por dichos compuestos si no sabe nada de ellos. Se encontrará con 4 botes que contienen N2, H2, NaOH y HCl, y deberá mezclarlos convenientemente para conseguir los compuestos solicitados por el Dr. Pipet. Al entregárselos, entrega a Ronin SO3, que lo tendrá que utilizar para conseguir ácido sulfúrico para deshacer el cofre que contiene el cuaderno que busca. Irá al despacho del profesor al por el cuaderno, y cuando le entregue todo lo requerido al Dr. Pipet, este prepara un pequeño examen para Ronin antes de volver a convertir a Harriet. ¡Tendrá 3 intentos para superar sus preguntas, o no volverá a ver a Harriet!

Objetos:

- N2, H2, NaOH y HCl: cada uno en su bote respectivo. El jugador tendrá que conseguir los compuestos que le ha pedido el Dr. Pipet a través de la

correcta combinación de estos elementos.

- Pila: es una pila típica de un laboratorio, para lavarse las manos. Necesitará recoger agua de ella en el cubo que recogió en los alrededores de la granja. Así, el agua se podrá combinar con el SO3 que le entregó el Dr. Pipet para conseguir ácido sulfúrico.

Personajes:

- Dr. Pipet: proporcionará los conocimientos necesarios para pasar sus pruebas en distintas conversaciones, a parte de la conversación final que consistirá en un examen. En la primera conversación que se mantiene con él ("drPipetConversation1") cuenta lo que ha pasado con Harriet y que debe hacer el jugador para que vuelva. La siguiente conversación que es posible mantener con él ("drPipetConversation2") es para preguntarle las nociones necesarias para conseguir los compuestos que solicita. Además, las conversaciones "drPipetConversationNH3" y "drPipetConversationNaCL" serán expresadas cuando se le entregue el amoniaco y la sal que pidió, respectivamente. Cuando entregue el SO3 se aplicará la conversación "drPipetConversationSO3" en la cual da una idea de que se puede hacer con este compuesto. Cuando el personaje principal entregue el cuaderno, aparecerá el examen, que es también una conversación ("Examen"). Por último, aparecerán las conversaciones que finalizan la aventura ("drPipetConversationNotepad", "endConversation" 1 y 2) en caso de que supere el examen. Si no lo supera en los tres intentos, tendrá otra conversación con el Dr. Pipet en la que nos indica que nunca verá a Harriet, acabando también la aventura (conversación "Final").
- Bestia: es en lo que se ha convertido Harriet tras el experimento.
- Harriet: aparece cuando el jugador entrega todo lo requerido al profesor y pasa el examen.

Salidas:

- La única salida que tiene el laboratorio comunica con el despacho del profesor, y no tienen ninguna condición.

Detalles de implementación:

- Conversación automática: Nada mas entrar en esta escena, el jugador mantiene una conversación con el Dr. Pipet. Esto es posible ya que añadimos en la acción de "Usar" la granja el efecto de lanzar conversación después del cambio de escena. Si hubiera sido un cuarto cerrado, se lo tendríamos que haber asociado a los post-efectos, pero en los entornos abiertos no tenemos otra manera.

- Implementación de Harriet: Para crear el efecto de que Harriet aparezca en la escena y desaparezca la bestia en el momento adecuado hemos asociado condiciones a ambos personajes. Poniendo a la bestia una condición (dicha condición hay que configurarla en el menú de escena del editor) de que aparezca en la escena si el flag "gameFinished" está inactivo, aparecerá en la escena, y haciendo lo mismo con Harriet, pero con la condición de que este activo, ella no aparecerá. Así, cuando pasemos el examen, activaremos dicho flag, apareciendo Harriet y desapareciendo la bestia.

- Implementación de las "mezclas": Para conseguir los compuestos que nos solicita el Dr. Pipet el jugador debe combinar los objetos que encuentra en la mesa de esta escena. Tiene que realizar la acción "Usar con" entre los objetos "H2" y "N2" para conseguir "NH3", y hacer lo mismo entre "NaOH" y "HCl" para conseguir la sal. Cualquier otra combinación entre estos elementos que no sea la descrita, generará un flag "BadMixID" (ID será los identificadores de los objetos involucrados en la mezcla), el cual nos servirá para la evaluación. De esta manera podemos evaluar si ha cometido errores el jugador a la hora de hacer las mezclas.

- Examen final: El examen final es una conversación de tipo grafo con el Dr. Pipet. En ella, después de cada pregunta llegamos a un nodo de opciones, del cual el jugador deberá elegir una (entre tres). Si elige la correcta, continua el examen, sino acaba. Si se equivoca antes de acertar cinco preguntas, el examen se considera suspenso, y si quedan intentos podrá volver a intentarlo mediante la opción "Hablar" (del HUD) con el Dr. Pipet. Si no quedan más intentos, al hablar con el doctor le dice a Ronin que no volverá a ver a Harriet y se acaba la aventura, por medio de un efecto de final de capítulo.

Si por el contrario el jugador se equivoca cuando ha contestado al menos 5 preguntas bien, el examen se considera como aprobado, por lo que si vuelve a hablar con el doctor, transformará a la bestia en Harriet y acabaría la aventura.

Cabe destacar que en el examen hay un conjunto de activaciones y desactivaciones de flags para el seguimiento del juego y para la evaluación. Cuando el jugador llega a 5 preguntas acertadas, si falla no se activa el flag "failExam", y al acertar activaremos el flag correspondiente al número de preguntas que lleva acertadas.

2.3 Mejoras educativas del juego

Libros de referencia

El jugador se encuentra con tres libros a lo largo de la aventura con información teórica sobre química. Además, estos libros le ayudarán a aprobar el examen final.

El primero de ellos aborda temas generales de la química, lo que viene a ser nociones básicas de la materia. El personaje principal puede obtenerlo en la conversación inicial con el profesor, a menos que cuando le pregunte sobre sus conocimientos de química, le diga que son elevados.

El segundo se encuentra al "Examinar" el cubo de los alrededores de la granja. En el se le introduce en el mundo de las reacciones químicas.

El tercero se lo entrega el Dr. Pipet cuando el jugador suspende por primera vez el examen. En este libro viene la información que ha sido transmitida en las

diversas conversaciones, como por ejemplo que hay que hacer para conseguir la sal o que ocurre cuando se mezcla el ácido sulfúrico con un metal. Por ello, este libro servirá al alumno para afianzar los conceptos, intentando que apruebe el examen.

Para conseguir que estos libros estén a disposición del alumno de forma permanente, se debe crear un objeto libro (uno por cada libro que queramos mantener) y asociarle un efecto que lance el libro cuando se ejecute la acción "Examinar" sobre dicho objeto.

Perfil de evaluación

Como hemos ido comentado a lo largo de este informe, podemos evaluar la acción del alumno en la aventura. Para ello se han añadido un conjunto de reglas de evaluación. Vamos a analizarlas:

- Mezcla incorrecta de metano y mechero: comprueba si el alumno ha intentado usar el mechero con el metano en su inventario. Se considera negativo ya que se ha insistido en lo explosivo de la acción.

- Las siguientes reglas comprueban si el alumno ha realizado alguna mezcla incorrecta con los componentes que le son dados:

  Mezcla incorrecta de H_2 con HCl

  Mezcla incorrecta de H_2 con NaCl

  Mezcla incorrecta de H_2 con NaOH

  Mezcla incorrecta de N_2 con NaCl

  Mezcla incorrecta de N_2 con NaOH

  Mezcla incorrecta de N_2 con HCl

  Mezcla incorrecta de NH_3 con HCl

  Mezcla incorrecta de NH_3 con NaOH

  Mezcla incorrecta de H_2SO_4 con el mechero

- Las siguientes reglas evalúan el examen:

  Examen suspendo

  Examen aprobado con 5 aciertos

  Examen aprobado con 6 aciertos

  Examen aprobado con 7 aciertos

  Examen aprobado con 8 aciertos

  Examen aprobado con 9 aciertos

  Examen aprobado con 10 aciertos

3. Conclusiones y análisis de resultados

Para desarrollar un videojuego se tienen que seguir 3 etapas claramente diferenciadas:

- Etapa 1 - Escribir el guión del juego: es básicamente lo que hacemos en el apartado dos de este capítulo. Habría que añadir también la trascripción de las conversaciones.
- Etapa 2 - Producción o adquisición de recursos artísticos: En nuestro caso, algunos recursos artísticos como las transparencias los hemos producido nosotros; pero los modelos en tres dimensiones han sido descargados de Internet.
- Etapa 3 - Desarrollo del juego: consiste en programar el juego para que sea ejecutable. En el caso de <e-Adventure3D> esta tarea es crear el archivo EA3D que contiene la aventura completa.

A continuación vamos a mostrar una tabla que relaciona el tiempo medido en días de trabajo (más o menos unas 6 horas por día) dedicado para cada una de las etapas mencionadas anteriormente para el desarrollo de la aventura de prueba con y sin el editor:

ETAPA	TIEMPO CON EL MOTOR	TIEMPO CON EL EDITOR
1	2	
2	3	
3	5	1

Al ser la misma aventura de prueba, las dos primeras etapas tienen la misma duración. Esto es debido a que al rehacer la aventura mantuvimos el guión del juego y sólo cambiamos algunos de recursos puntuales (como añadir transparencias y cambiar el modelo tridimensional del personaje principal del juego). En encontrar los recursos adecuados tardamos tres días debido a que, en su mayoría, los descargamos de páginas gratuitas de internet. Lo difícil es encontrar el modelo tridimensional adecuado a lo que se tenía pensado en el guión del juego. De todas maneras, este no es el punto a tratar en este apartado; ya que, los modelos se pueden comprar o desarrollar. La etapa que aquí nos incumbe es la tercera. Como podemos ver, en desarrollar la misma aventura de prueba tardamos 5 veces menos utilizando el editor (incluso añadiendo 'assessment' y libros que en la aventura inicial no se utilizaban). Esto es debido a las aportaciones del editor con respecto al motor que ya mencionábamos en el apartado de 'características de <e-Adventure3D>'.

Para que esto fuese un caso de estudio completo, faltaría que algunos alumnos probasen el videojuego y analizáramos los resultados. Es decir, analizar el valor educativo del juego, ya que, al fin y al cabo, este es el fin que tiene el proyecto (además, lógicamente, del desarrollo de juegos con un coste bajo).

Como conclusión, nos gustaría destacar que el desarrollo de un juego en tres dimensiones desde cero es una labor que, generalmente, involucra varios programadores y muchas horas de trabajo. Con este caso de estudio queda demostrado que en <e-Adventure3D> la labor de desarrollo se convierte en una tarea sencilla que no requiere ningún conocimiento de programación. Esto hace que un profesor cualquiera pueda elaborar el guión de un juego educativo y desarrollarlo fácil y rápidamente usando el editor con tan solo adquirir los recursos artísticos necesarios.

Chapter XIII

Conclusions

1. Discussion of the results and goals achieved

In our opinion, <e-Adventure3D> has been a total success. Not only we have achieved the main goals proposed, but also the project has reached limits much beyond our imagination at the beginning of the project.

In reference to the first objective set on section III.3:

> **(1)** *Produce an environment authoring platform for the creation and execution of 3D educational adventure videogames with no need of programming or game-making skills with a low production cost.*

The goal is fully achieved. Let's analyze it step by step.

Firstly, as it is obvious the editor tool allows users to create complete 3D adventure games. All the common elements of adventure games have been extrapolated and now can be easily edited. Between these elements we can find the scenes (interactive scenarios), cut-scenes (multimedia non-interactive scenarios), objects, characters, the player, a full interaction system based on actions and conversations (which do not differ in complexity of what you can do with the conversations you can find in titles such as *Escape from Monkey Island* ® or *Grim Fandango* ®), a full system to set up the narrative course (flags) and a complete list of effects to give games more expressivity. Summing up all, the platform supports the creation of adventure 3D videogames because no essential elements of the genre are missing (of course more elements would need to be added to enhance the potential of the games).

Secondly, it is true that there is no need to have programming or game-making knowledge. Even we have gone further: the game creators will only need basic computer formation, as the editor supports the edition and creation of all the features of the games so there is no need to know the <e-Adventure3D> language.

Finally we move to the cost of the games. The most important challenge of the project was that the platform should reduce the development costs enough to be affordable in educational contexts. Well, we think the challenge has been overcome: the games produced are really low cost games. In our opinion it is amazing that complete functional 3D games could be created with just a few mouse clicks. As the two test games developed prove (tech demo and case study), with an effort of 1 or 2 person-week it is possible to develop an educational 3D adventure

game. Let's analyze for instance the development costs of the tech demo. The game was developed to show the characteristics of the engine (i.e. what the engine can do), and finally got quite complex. It is composed of four chapters and the average play time is of 25-30 minutes. The total XML lines ascended up to 2390 (534, 644, 657, 555 for each chapter respectively). The total size in disk is around 150Mb. The next table shows the development costs broken down in tasks.

Task	Time spent
1. Creation of the storyboard (4 chapters)	**5h**
2. Production and search of the art assets	**9.5h**
2.1. Search of the 3D models	3h
2.2. Creation of the cut-scenes	1h
2.3. Creation of the dialog sounds and background Music	4h
2.4. Search of skyboxes and creation of textures	1.5h
3. Implementation of the game with the editor (4 chapters)	**9h**
4. Testing and improvements	**6h**
Total	**29.5h**

As it can be observed, to develop the game only took 30 hours of work (approximately a person-week). It is true that the tool was managed by people instructed to use it, and that the most expensive art assets, the 3D models, were took from web sites which store free 3D models for download. Even taking these considerations into account, the costs of the implementation are really low. The costs of the art assets will depend on the game (however here it is proved how a full game can be developed without expending great efforts on the production of the assets).

Concerning the second objective:

(2) *Endow the platform with features for the assessment and adaptation of the learning experience. The platform must present the ability to*

> *communicate with Learning Management Systems so the evaluation produced can be attached to the profile of the student and the adaptation is carried out according to it.*

The platform can communicate to a LMS. The platform has been doted with mechanisms to take advantage of the interaction game-player for the automatic evaluation of the student and adaptation of the game. Both assessment and adaptation rules are expressed in terms of flags, so they are related to the concept of game state in <e-Adventure3D>. Nonetheless, it is true that those rules are perhaps too simple. More complex rules (above all for assessment) are required. For instance, it would be nice to support timers so the time needed by the user to complete a certain task could be used for assessment purposes. However, we think the objective has been achieved, but we still think that more research is needed in these fields.

Finally, we have to talk about the adventure editor. At the beginning of the project it was complicated to design an application which could be used for the development of adventure games by instructors. Good 3D game makers are usually complex utilities which are in the order of hundreds thousand source code lines. Such kind of application would have been impossible to be developed by three people in 10 months of work. On the other hand, the editor should be easy to use so instructors could use it. Therefore we only planned to develop an editor to help in the edition of those features too complicated to be written directly using the XML language, such as the location of the elements in the scenes. Nonetheless, the results defied our imagination. Now the editor is a rich tool which can be used to produce full games with no need to know about the <e-Adventure> language. As long as the development of the tool advanced we were able to visualize how to address issues unreachable at the beginning, such as a simple method for the location of the elements which could be implemented in a short time. Finally the editor has become the key part of the project, to the detriment engine. That is what really reduces the development costs of the project.

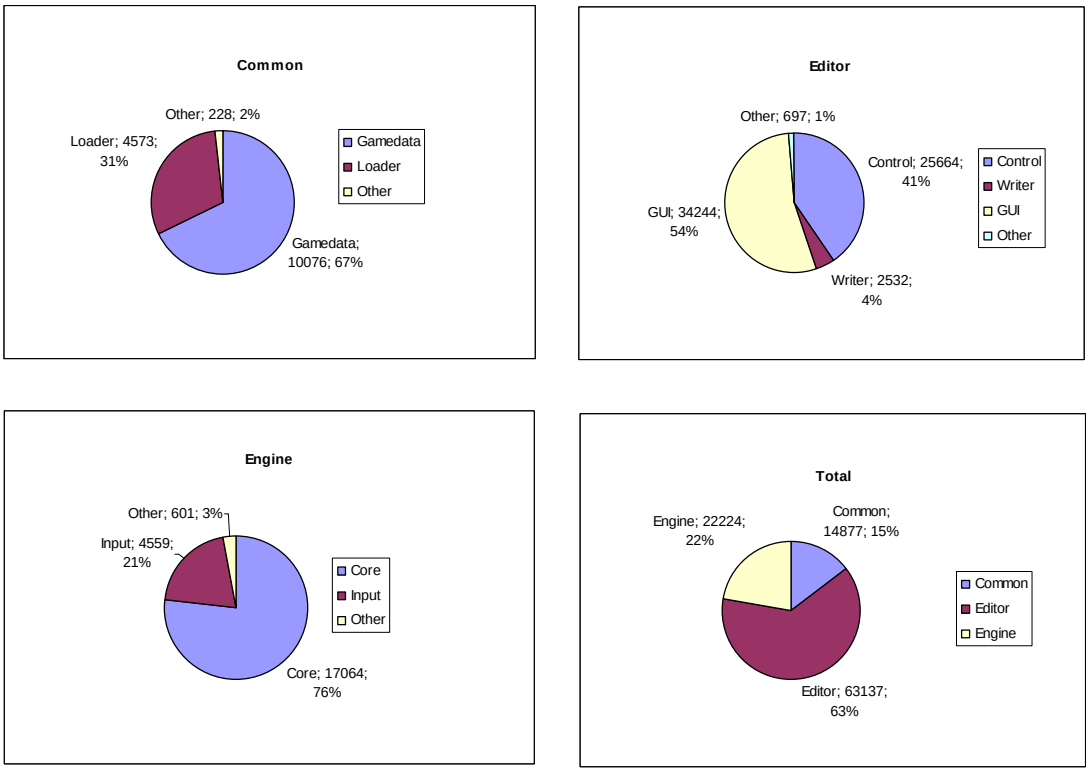
2. Analysis of the final complexity

From the beginning it was determined that the complexity of the project was high. 3D games development is always a costly task, incremented by our

inexperience in the matter. However, we could never imagine the dimensions it finally got. As the next table shows, the total source code lines of the project raised up to more than 100 000 lines and 599 classes! The editor application, which at the moment of its conception was not more than a simple tool, has become the most costly and complex application of the project with a total of 63137 lines and 331 classes, without taking into account the common classes (14877 lines and 119 classes) shared between editor and engine. Finally, the engine is compounded by a total of 22224 SOCL and 149 classes.

Unit	Nominal Lines	Source Code Lines	Comment Lines	Blank Lines	Mixed Lines	Classes
Common	**18625**	**14877**	**704**	**3031**	**13**	**119**
Gamedata	12184	10076	117	1990	1	94
Loader	6169	4573	584	1000	12	21
Other	272	228	3	41	0	4
Editor	**78706**	**63137**	**4472**	**10867**	**230**	**331**
Control	31509	25664	1536	4148	161	99
Writer	3382	2532	382	459	9	17
GUI	42965	34244	2525	6137	59	211
Other	850	697	29	123	1	4
Engine	**28538**	**22224**	**1138**	**5135**	**41**	**149**
Core	21639	17064	943	3595	37	104
Input	6140	4559	160	1417	4	42
Other	759	601	35	123	0	3
Total:	**125869**	**100238**	**6314**	**19033**	**284**	**599**

Source code lines counted using Practiline Source Counter Pro trial version.



And all of this taking into account that we have used a free 3D engine which help us with most of the common features in 3D game development. That gives a good idea of the real complexity of the project. And all these work was not easy. During the development of the editor and engine there was lots of complications we had to solve. Most of them were related to our inexperience in game development and due to the short life of the JME (our 3D engine). No documentation is available, so in most of the cases we had to access directly to the native code of the application for it customization, along with the source code of the java libraries to adjust thing such as the layouts, file choosers, etc.

3. Discussion of the learning value of the games

We would like to spend a bit of time to analyze the educative value of the games produced with the platform. It is true that this is a feature difficult to measure theoretically as experimentation in real courses is needed. However after have developed two test games we think <e-Adventure3D> games can teach. The key to achieve the learning value it is not in the platform but in the storyboard (as it happens in commercial industry where adventure games success due to a good script).

That is the platform presents the mechanisms and features to create games with a high educational value but the responsibility of the final achievement concern exclusively to the game producers (otherwise the introduction of the contents by

instructors would remain an impossible mission). Just as a brief remainder, the mechanisms we are talking about are the possibility to provide information (in-game books), evaluate the student, adapt the contents according to the user profile, and connect the games to the LMS so the learning experience is not disconnected from the curricula, or the possibilities contributed by the multimedia content shaped as cut-scenes. Nonetheless those not education-specific features, such as lights, cameras or conversations will do their bit for the educational value as they can be used to guide the student, provide information, etc.

4. Future work

Although in our opinion the project was a real success, there is still being a lot of work to be done. The possibilities of both editor and engine are enormous due in part to the potential of the JME Engine (which is still in an early development stage and will add new features soon). In this section we analyze the future work that can be done with it split in two headers: issues that need a further investigation so the work goes on in the line of getting costs reduced and involving the instructors in the process, and the features that can be added or improved in the platform to enhance the quality of the product.

3.1. Research issues

Although the project contributes considerably to the integration of educational videogames in the educative system, there is still research to be done. Here we discuss some of those issues.

Reduce the production costs of the art assets.

The problem of the high development cost has been reduced enough so educational budgets could afford it, but it is not enough in some situations. Educational or public organizations such as colleges, universities or state education departments could afford the production of some games, but it is still elusive for high school teachers which do not have at their disposal a team to produce the art assets, and which in few cases will be able to spend a whole week developing the game. In that situations the cost of the art assets, specially the 3D models, are a bottle neck which will deserve further attention. Nevertheless it is not an issue in which we could contribute as we are not art designers.

However there is a simple solution to tackle this issue: collect and produce a complete library of art assets to be distributed along with the platform. There are cases in which the models and other assets of the game need to be produced

explicitly, but for low cost simulations and cases in which the realism is not so important a good collection of models will serve. This task will not be very hard as there are plenty of web repositories which store free static models (the only work there is to gather them together). The problem comes to obtain animated models, which are not so easy to find. That could be solved thanks to a small utility for the customization of pre-defined characters (as most of commercial videogames include) embedded in the editor.

Deployment of real study cases in real courses.

Another point that would need some effort is to test the real effectiveness of the games produced with <e-Adventure3D> in real courses. This experimentation stage is mandatory in research projects, but it is usually a time-consuming task and due to lack of time we could not carry it out. In our case the experimentation would be focused on testing the efficacy of the learning experience provided by the games (i.e. if the students really learn with them).

3.2. Enhancing the quality of the platform

The editor tool allows anyone with no knowledge about <e-Adventure3D> language or XML to create his own educational games even if they are not experts in computers. In addition, the engine provides an intuitive way of playing the games that provides the player a fun and educational experience as it was supposed to be. It can be seen that the project is a very complete platform at its own; however there are some paths to follow in order to extend and improve its characteristics:

- To take advantage of the JME potential: There are some characteristics JME provides and we have not taken advantage of because there was no time to explore everything. One of them is the management of animations for models MD2 and MD3. We do not support them in the platform because these animations are controlled with the 'key frame controller' and we use the 'joint controller' which is the most popular. It is used for models such as MS3D ('Milkshape' models). Models with animations are expensive and we have just five MS3D models, so that we can not affirm that every animated model will work.
- Editor help assistant: It would be very useful to use any kind of virtual assistant to help users with the editor tool while they are developing an adventure. For example, when a user creates a new scene the assistant will tell him what kind of things he can add to the scene or how to change some parameters that define the scene environment.

- Engine help: It would be also helpful to do a similar improvement in the engine. For example, when an adventure is executed for the first time, give some advises to the player. It is a very popular practice in commercial games. At the beginning of a game it could be used to tell users what the inventory is used for or which game pad key users must press in order to grab objects (without the necessity of reading a tutorial)
- Add a scripting language to the platform: People with programming background could improve the expressivity of their games. This is a much advantaged characteristic that would require a great amount of work.
- Improvements in 'assessment' and 'adaptation': Firstly, we propose to extend the launch of assessment rules so they could be launched at any time of the adventure (with a timer or something like that). It could improve the automatic evaluation of students. Moreover, another extension could be to let the student to write answers to questions so they would be shown in the assessment reports. Secondly, we propose an extension of the LMS server. The current LMS server used for assessment and adaptation was developed by the department and it would be very useful to extend it with another layer in order to allow a standard communication with the server (for example, using the standard SCORM).
- Allow the use of timers in the effects system: This would be a very useful improvement. If the use of timers is allowed, effects would be launched whenever the game maker wants and not only after a player action.
- Improve the book rendering: We have thought to develop another type of books. Books that show in their pages HTML webs (including the use of flash technology). It will enrich the books quit a lot.
- Improve the slides scenes: The slides scenes could be extended allowing the use of power point presentations. Obviously this will turn presentations in something more complex because all the characteristics of power point.
- Text-to-speech conversations: Now conversations allow sounds in each conversation line, but these sounds must be recorded by the game maker. We propose to include a new characteristic that transform text to a sound file automatically (for example, using something similar to java speech or voice xml). It will reduce a lot the game development costs (mainly time costs).

- Allow the exportation of videogames as Learning Objects (LO) which could be interpreted and deployed by common LMS (i.e. Moodle, WebCT, .LRN platforms, etc.): This would be related to the packing of the games following the IMS-CP specification and the attachment of standard metadata as the IEEE LOM (Learning Object Metadata) specifies.
- Improve the game pad configuration interface: New game pads can be configured with our platform in order to play <e-Adventure3D> games. However, it does not have a very intuitive interface.
- Extend the 'game state' concept: The game state of <e-Adventure3D> is defined by the flags states in each moment of the game. Something very useful would be to extend the game state concept with numerical variables. New effects would have to change the value of these variables and new conditions like 'greater than' or 'less that' should be added. It would be very useful to evaluate students because a variable could be kept with the number of tries of doing something like, for example, the number of attempts the student fails a question.
- Translate the editor tool to another language: It is a very easy task because we parse every single word written in the editor tool from an external XML file. It would be just translate it and change the references to this file.

Chapter XIV

Appendices

A. Editor Tool user manual

1. Comenzando

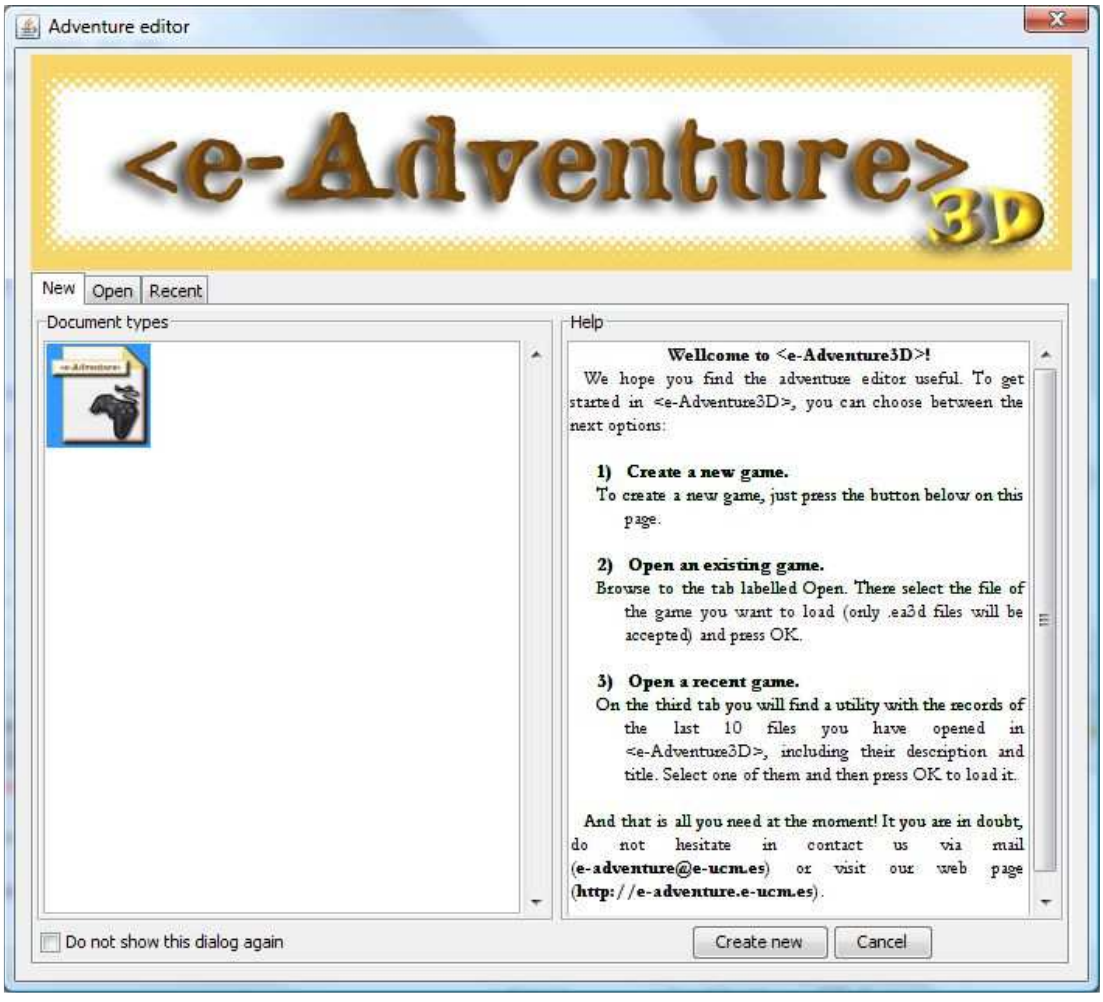
1.1. Respecto a este documento

Bienvenido a la plataforma <e-Adventure3D>. En este tutorial aprenderás a producir y ejecutar un juego funcional completo usando el editor de aventuras 3D.

Con este documento pretendemos que un usuario no familiarizado con la plataforma pueda desarrollar una aventura de principio a fin.

1.2. Toma de contacto

Una vez que tenemos una idea inicial sobre la plataforma, es el momento de empezar. Lo primero de todo es tomar contacto con el editor. Una vez que hayas ejecutado la aplicación jar, verás el diálogo siguiente:



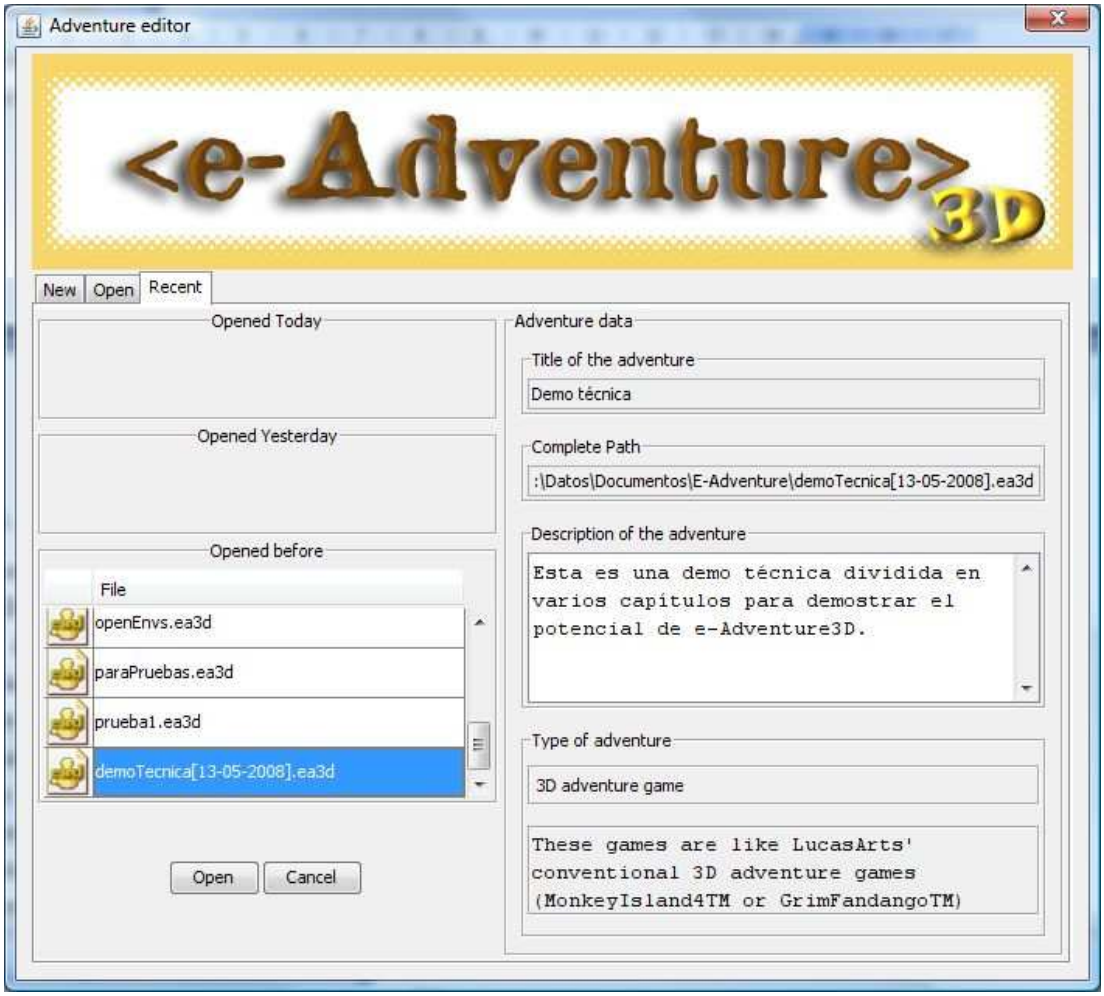
Este diálogo nos permite elegir entre las opciones iniciales, ya sea empezar una aventura desde el principio, continuar una que tengamos guardada o abrir una de las que se han abierto recientemente:

Crear un nuevo archivo.

Nada mas abrir el editor, la primera de las opciones que podemos elegir es la de crear una aventura desde el principio. Simplemente tendremos que pulsar el botón "create new" para empezar el proceso de desarrollo de una nueva aventura.

Cargar un archivo.

Seleccionando la pestaña "open" nos dará la opción de buscar en nuestro ordenador un archivo de extensión .ea3d (la extensión que utiliza la plataforma <e-Adventure3D>).

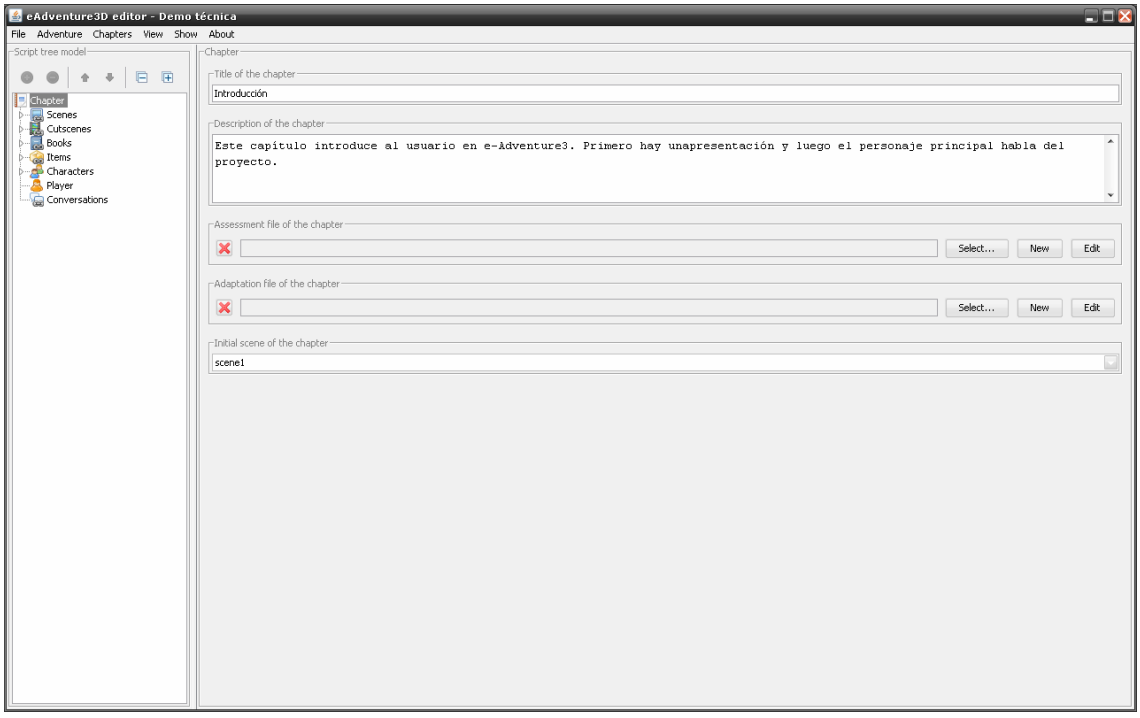


Archivos recientes.

Seleccionando la pestaña “recent” nos mostrará las aventuras que han sido cargadas recientemente. Nos indica las aventuras abiertas hoy, ayer, y con anterioridad, mostrándonos el archivo en cuestión, su título, descripción y ruta.

2. Creando mi primer juego <e-Adventure3D>.

Una vez hayas abierto un archivo o hayas creado uno nuevo, lo primero que verás será la pantalla principal de edición.



El editor de aventuras esta organizado siguiendo una estructura simple. En la izquierda esta el árbol de datos, utilizado para indexar todos los tipos de elementos que pueden ser utilizados en una aventura. Cuando hacemos clic en un nodo del árbol podemos ver a la derecha todas las opciones que nos permite editar, y los hijos de ese nodo son los elementos concretos de este tipo que se han creado en la aventura. De todas maneras se puede utilizar el árbol para añadir, borrar, acceder y organizar los elementos de la aventura. Cuando seleccionamos un elemento se puede editar en el panel de la derecha. Encima de ambos paneles encontramos la

barra de menú en la cual podemos editar otros aspectos del juego que serán posteriormente explicados. Primero, vamos a explicar los elementos que forman una aventura de <e-Adventure3D>.

2.1. Capítulos

Un juego <e-Adventure3D> está organizado en capítulos. De esta manera el juego está fragmentado en pequeñas piezas fáciles de diseñar, editar y guardar en memoria cuando se ejecuten. En cada capítulo está auto contenido un mini-juego. Todos los elementos definidos en un capítulo no son accesibles desde otros.

En el editor, solo se edita un capítulo al mismo tiempo. El menú de "capítulo", localizado en la barra de menú, nos permite añadir y eliminar capítulos, y observar el capítulo que queremos editar. A través de este menú podemos gestionar los flags del capítulo actual, pero esto lo discutiremos mas adelante.

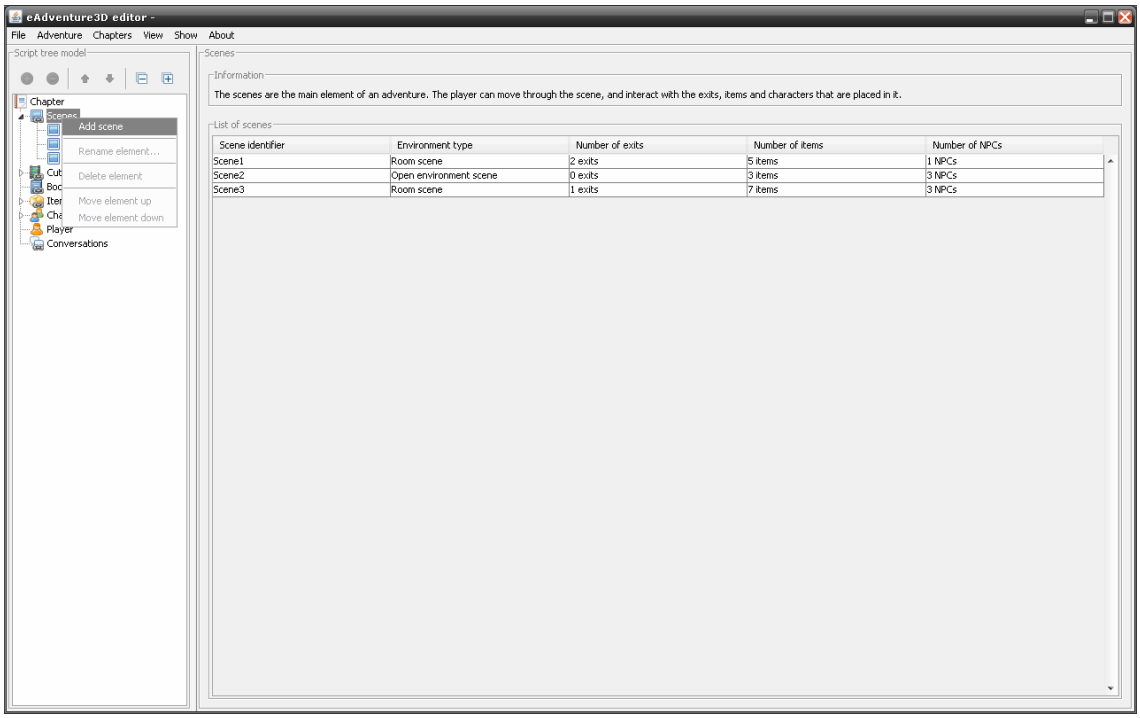
Cuando un capítulo está seleccionado, podemos editar su título, su descripción, y la escena inicial del capítulo. Además, los ficheros de adaptación y evaluación del capítulo pueden ser gestionados aquí. Aprenderemos a realizar estas acciones cuando lleguemos a la sección llamada "Adaptación y Evaluación".

2.2. Escenas

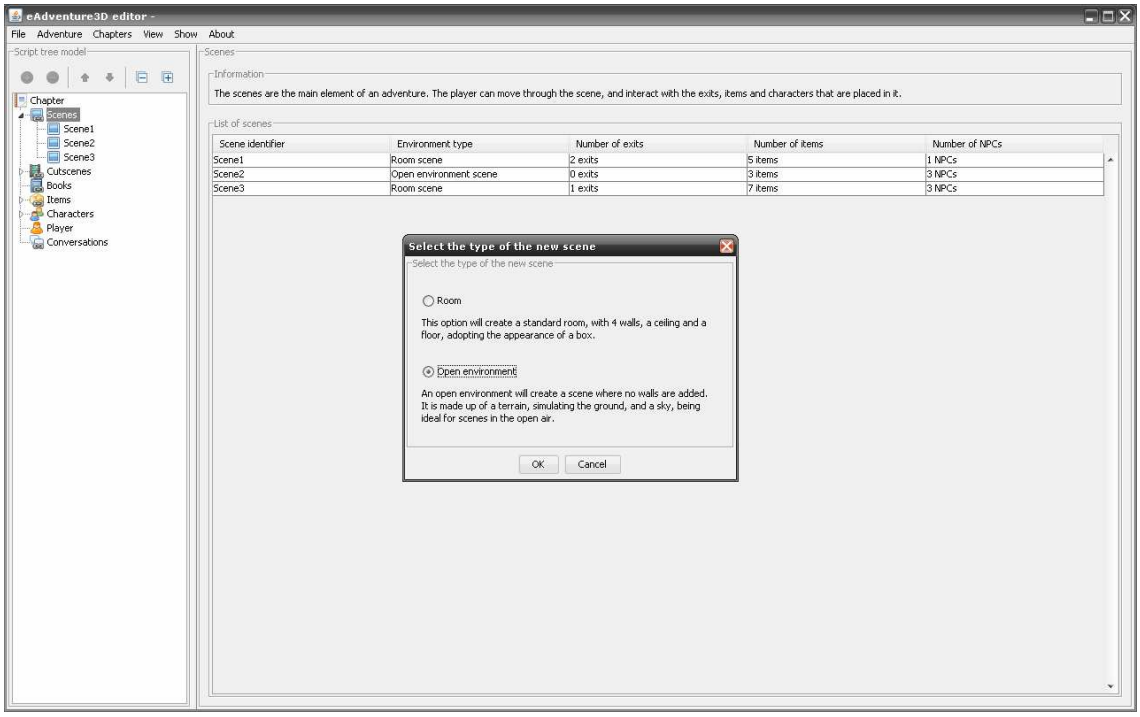
Cada capítulo está organizado en escenas. Hay dos tipos principales de escenas: escenas convencionales y "cut scenes" (en español llamadas escenas de corte, como las denominaremos de ahora en adelante). Las escenas convencionales son escenarios donde el jugador interactúa con los objetos y personajes, y están relacionadas con otras escenas mediante salidas. Por otro lado, las escenas de corte son vehículos para aumentar valores educacionales de la aventura. Fundamentalmente hay dos tipos de escenas de corte: "slide-scene" o diapositivas y "video-scenes" o escenas de video. Las primeras son una sucesión de imágenes, mostradas a pantalla completa una detrás de otra. Una escena de video es, como su nombre indica, un video reproducido en pantalla completa. En esta sección aprenderemos a crear nuevas escenas y escenas de corte.

Añadir una escena

Primero, debemos hacer clic con el botón derecho del ratón en el nodo llamado “Scenes” o “Cutscenes” en el árbol de la izquierda, dependiendo del tipo de escena que queramos crear, tal como muestra la siguiente figura:



Pinchando sobre “scenes” en el árbol, nos aparecen las escenas existentes en el capítulo, así como el tipo de entorno, y el número de salidas, objetos y personajes no principales. Comenzamos creando una escena convencional nueva. Dentro de <e-Adventure3D> tenemos dos tipos de escenas convencionales: un entorno abierto, en la que no encontramos paredes, sino un terreno y cielo; un cuarto, en la que nos encontramos con una escena con cuatro paredes, techo y suelo. Una vez creada, procedemos a editarla.



Empezaremos con la edición de un cuarto (las escenas de corte las veremos en la sección 2.7). Eligiendo esta opción nos aparecerán las siguientes pestañas:

Cuartos

"Pestaña Locator"

Aquí podemos ver una previsualización de como quedará la escena. En la parte inferior tenemos tanto la edición y creación de cámaras como la de luces. En la parte derecha nos encontramos con las pestañas que nos permiten gestionar los objetos, personajes y regiones que aparecen en la escena, así como la posición inicial del protagonista.

Cámaras

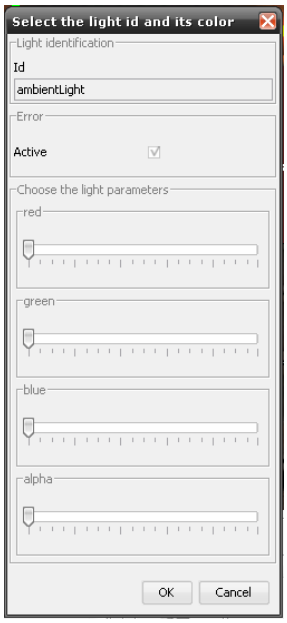
Podemos ver las cámaras que tenemos para la escena. Por defecto, nos sale una cámara estática. El primer icono nos permite crear una cámara estática en la posición en la que nos encontramos en la ventana de previsualización, pidiéndonos el nombre que tendrá dicha cámara. El siguiente icono, borra la cámara que tenemos actualmente seleccionada. El tercer icono en discordia nos permite crear una cámara en tercera persona, es decir, que seguirá al personaje principal. Al

pinchar sobre este icono, nos pide que dejemos la inclinación y la distancia a la que estará la cámara del personaje principal, para ello basta con mover el ratón y la rueda del mismo. Pulsamos el botón de aceptar cuando tengamos elegida la posición, y nos pedirá el nombre que queremos para dicha cámara.

Una vez creadas las cámaras podemos elegir cual será la cámara por defecto seleccionándola y pinchando en "set as default". Si pinchamos sobre cualquiera de las cámaras creadas, automáticamente se colocará esa cámara en la previsualización.

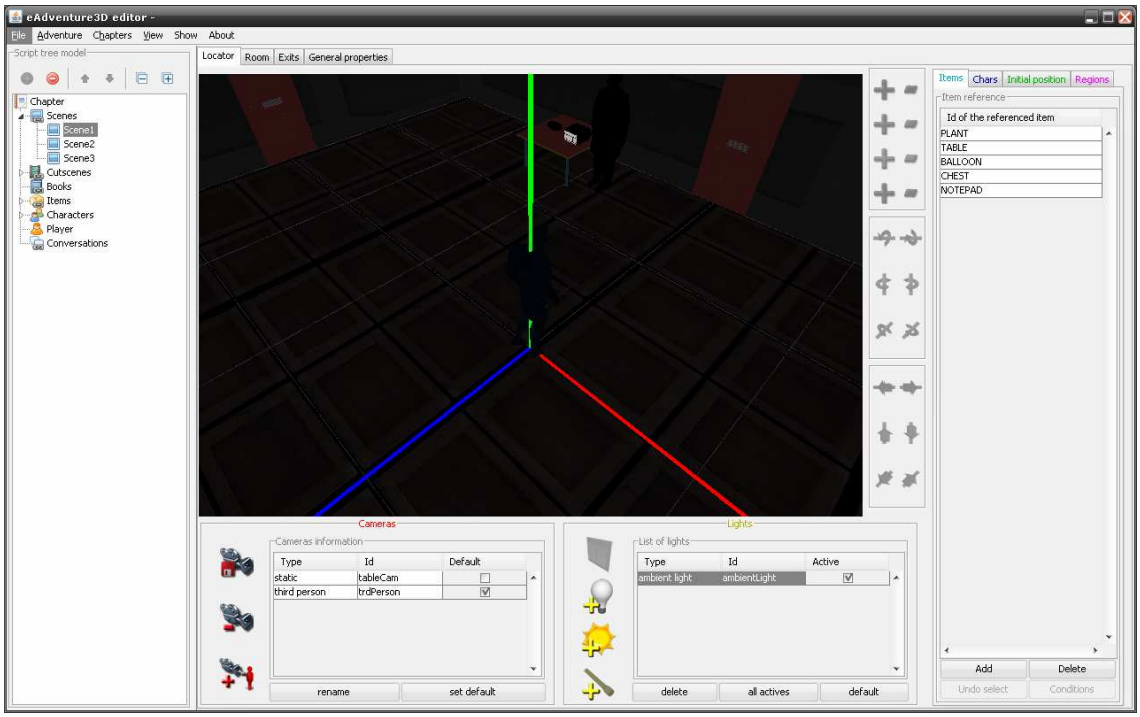
Luces

Al igual que con cámaras, podemos definir distintos tipos de luces en una escena. Nada mas crear la escena automáticamente se le da una luz por defecto que ilumina por igual toda la escena. Esto se hace para que las luces sean totalmente opcionales a la hora de crear una aventura; pero siempre se pueden crear nuestras propias luces en la escena. Al principio, el único tipo de luces que se nos permite crear es el de "luz de ambiente" pinchando en el primer icono del recuadro (ya que tiene que existir la luz ambiente para que existan los demás tipos de luces). Al seleccionarlo nos aparece el siguiente diálogo:

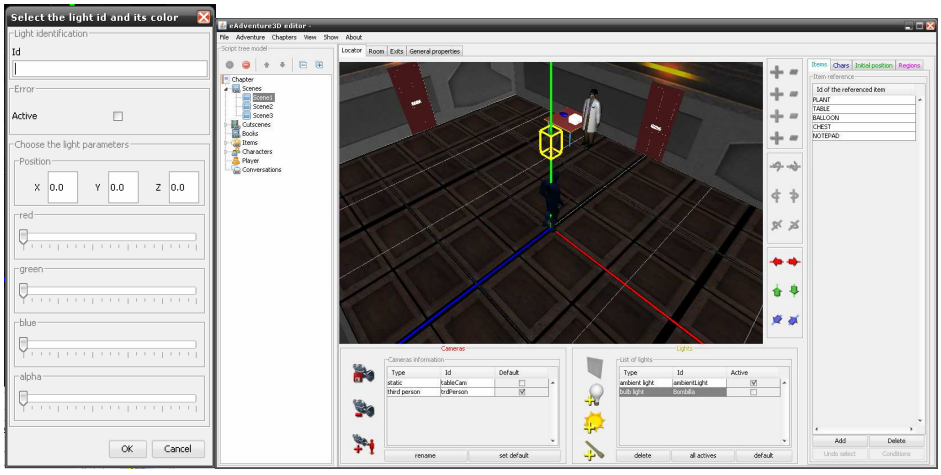


En él podemos elegir los parámetros que determinan el color ("red", "green", "blue") y opacidad ("alpha"). Para el resto de luces se nos permitirá escoger un

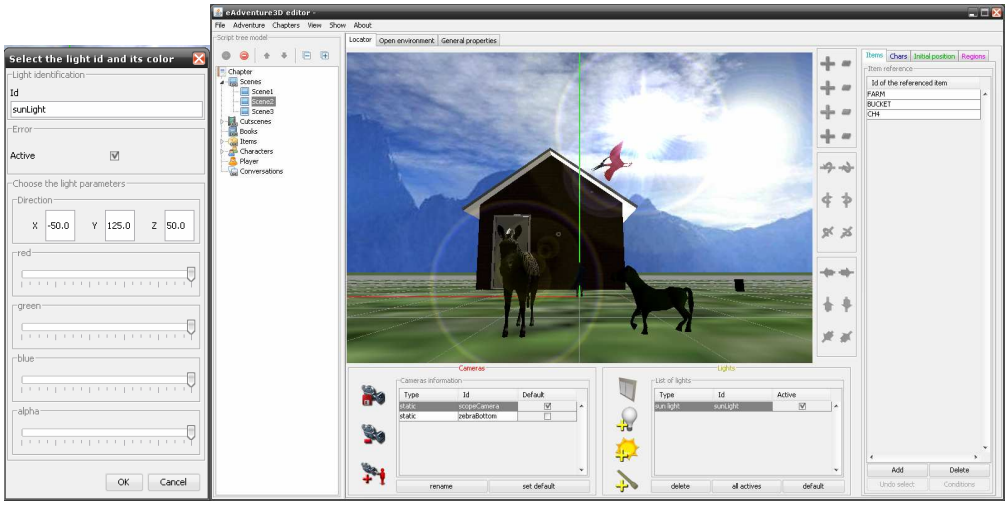
identificador y decir si esa luz está activa desde el principio o no. Vemos como queda la escena después de elegir una luz ambiente con todos sus parámetros de color a uno (a la derecha del todo en las barras):



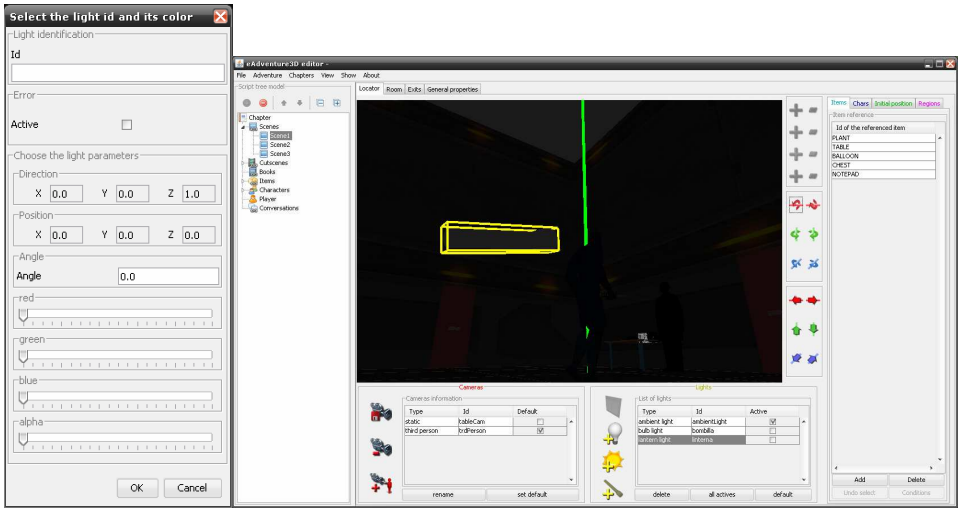
Como vemos, la luz ambiente sólo hace que los objetos se vean; pero no es suficiente. Una vez creada una luz ambiente, podemos crear luces de cualquier otro tipo. Las luces de tipo bombilla nos dan una iluminación uniforme. Debemos, además de rellenar los parámetros anteriores, colocarla en la escena, ya sea dando su posición o colocándola con los iconos de desplazamiento (explicado en la sección 2.3.2). Este tipo de luces nos son útiles para alumbrar un cuarto por igual. El modelo tridimensional que representa la bombilla que aparece ahora en la escena es orientativo para poder colocarla en la escena, en la ejecución del juego (en el motor) no se dibujará:



Las luces de tipo sol nos son más útiles para entornos abiertos. La particularidad de este tipo de luces es que aparece la luz en la escena y hace un efecto parecido al del sol. Para colocar este tipo de luz, sólo lo podemos hacer a través de las coordenadas que aparecen en el diálogo, y no podemos utilizar los iconos de desplazamiento, ya que en realidad estamos indicando la dirección de los rayos del sol y no la posición.



Las luces de tipo linterna son como un foco que apunta en una dirección. De esta manera, a parte de la posición debemos elegir una dirección a la que apuntan y un ángulo. Este tipo de luces nos son útiles para alumbrar a objetos y personajes y destacar su presencia en la escena. Una vez más, la linterna que aparece en la previsualización es orientativa, no aparecerá durante la ejecución del juego aunque si el efecto que produce:



Una vez tenemos creadas todas las luces que nos interesan, vamos a ver como seleccionarlas según diferentes situaciones. En las que tengamos el checkbox

"active" seleccionado son las que estarán activas al iniciarse la escena. Como hemos dicho la luz ambiente siempre estará seleccionada. Podemos crear efectos de cambios de luces para activar y desactivar las luces creadas en una escena.

Definir la posición inicial del jugador en la escena

También tenemos la posibilidad de decidir donde aparecerá el personaje principal en la escena nada mas entremos en ella, lo que viene a ser la posición inicial. Para ambos tipos de escena, en la pestaña "Locator" nos aparece a la derecha la pestaña "Inicial position". Si seleccionamos el botón "Edit inicial position", se selecciona el jugador principal y podemos moverlos con los iconos de de desplazamiento (como se verá en la sección 2.3). Una vez hemos colocado el jugador en la posición elegida, pinchamos en el botón "Stop editing" para almacenar esa posición como la inicial para esa escena. Podemos poner la posición inicial o (es decir, en donde aparecen los ejes) pulsando el botón "Reset inicial position".

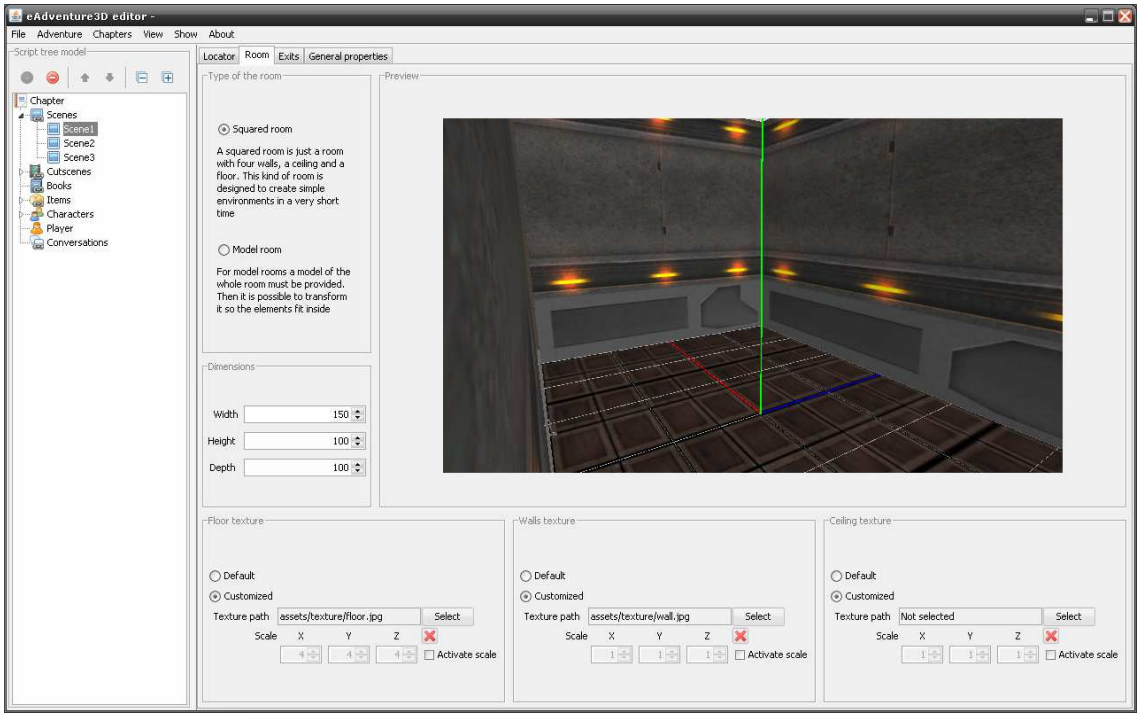
"Pestaña Room"

En este apartado vamos a gestionar los parámetros del cuarto. Tenemos dos tipos de cuartos (los cuales se pueden elegir en el recuadro "Type of the room"): Los cuartos convencionales ("Squared room") son habitaciones con cuatro paredes, un techo y un suelo; para los modelos de cuarto ("Models room") utilizamos un modelo de un cuarto para la creación de una escena de este tipo.

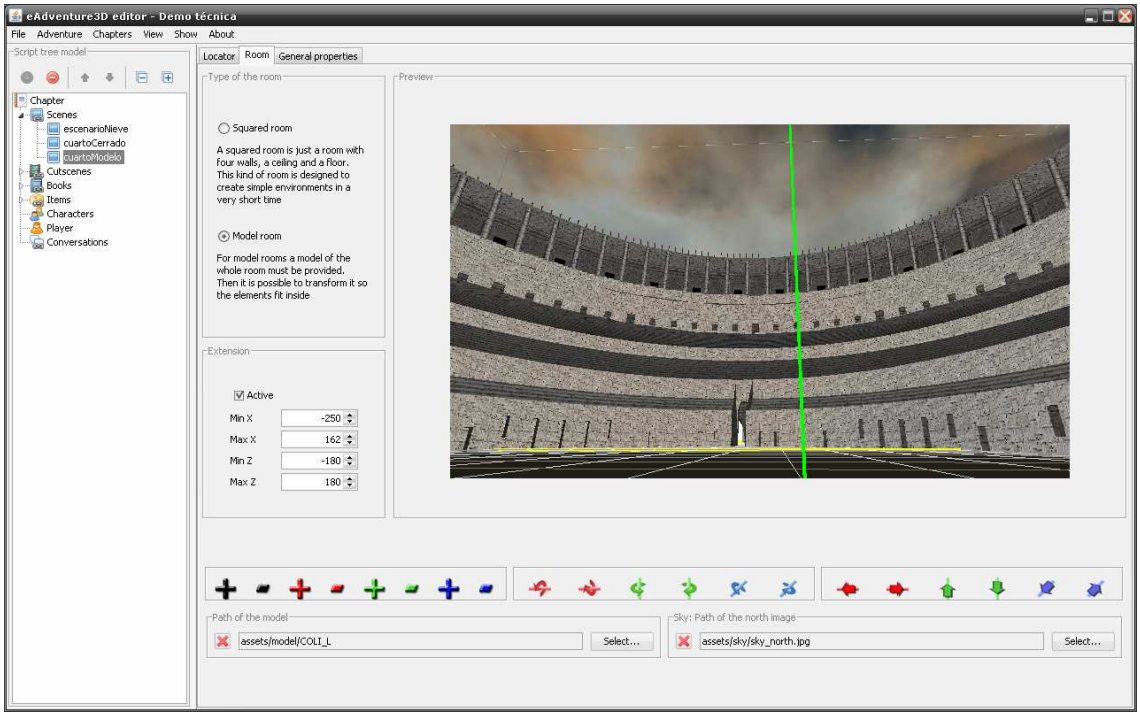
Para los cuartos convencionales podemos elegir los valores de anchura, altura y profundidad que va a tener nuestro cuarto.

Además también podemos elegir las texturas del techo, suelo y paredes. Nos vienen unas dadas por defecto.

Todos estos cambios se ven reflejados en la ventana de previsualización.



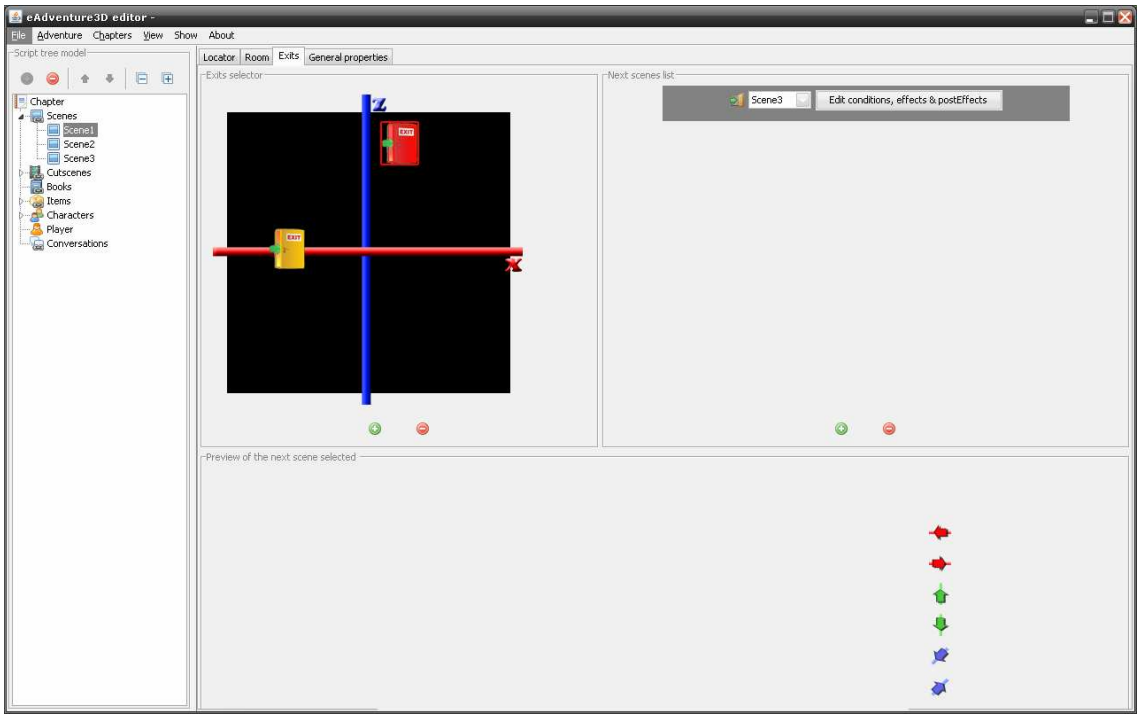
Para los cuartos creados a partir de un modelo podemos gestionar otros parámetros distintos a los de los cuartos cerrados. En el recuadro “Extension”, si lo tenemos activo, nos permitirá delimitar los valores máximos y mínimos que tendrá el espacio en el que se puede mover el personaje en el plano XZ (el suelo). Además nos aparecen los iconos de escalado, rotación y desplazamiento del modelo, los cuales explicaremos con mas detalle en la sección 2.3. En la parte inferior nos encontramos con los recuadros “Path of the model”, en donde deberemos elegir la ruta del modelo de cuarto, y “Sky: path of the north image”, en el cual podemos elegir la ruta de un sky box por si queremos que haya cielo para completar el modelo en cuestión.



“Pestaña Exits”

Las salidas sólo se utilizan en los cuartos cerrados, no en los creados a partir de un modelo. Aquí nos vamos a encargar de decidir donde habrá salida en el cuarto, la cual será representada por una puerta. Para crear una salida nueva, debemos hacer clic en el icono del recuadro “Exit selector”, en donde nos aparecerá una puerta. Seleccionamos la puerta recién creada, y nos vamos al recuadro “Next scene list” para añadirle siguientes escenas a esta puerta, eligiendo para la siguiente escena una de las creadas en el capítulo (nos aparece un botón que tendrá una lista con estas escenas). Podemos añadir más de una “siguiente escena”, y añadirle a cada una un conjunto de condiciones (botón “Edit conditions, effects and posteffects”). Iremos a la escena para la que se cumplan todas las condiciones, no pudiendo salir del cuarto si no llegamos a cumplir las condiciones de ninguna escena. Además podemos añadir efectos y postefectos, los cuales se ejecutarán antes y después de salir de la escena. Todo esto (condiciones y efectos) se explicará con más detalle en la sección 3. También podemos elegir que la salida en cuestión sea final del capítulo, por lo que al usar dicha salida, y si se cumplen las condiciones, acabará el capítulo en cuestión.

Una vez hemos elegido una escena nos aparece una previsualización en el recuadro “Preview of the next scene selected”.



"Pestaña General properties"

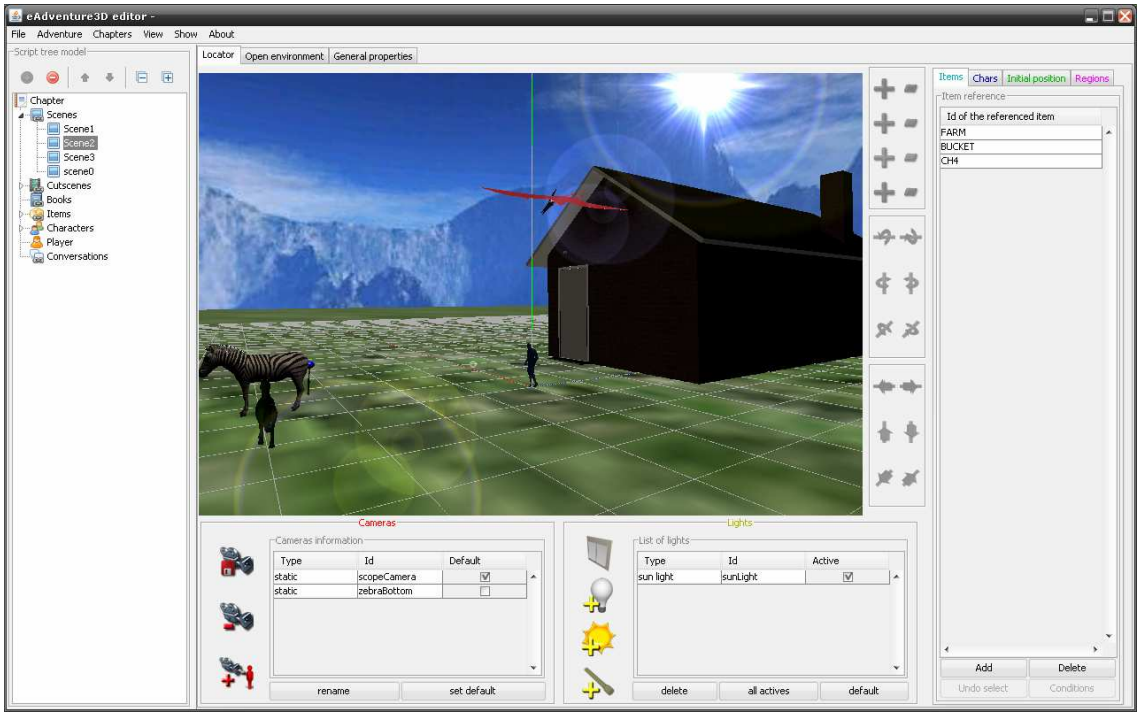
En esta pestaña podemos añadir información complementaria a la escena (documentación). Además de poder cambiar el nombre de la escena. A parte de esto, podemos elegir una música de fondo para la escena, eligiendo la ruta del archivo de audio (de tipo ".wav" u ".ogg") en el recuadro "Background sound of the scene".

Entornos abiertos.

Aquí vamos a gestionar los parámetros de configuración del entorno abierto, los cuales se nos irán mostrando en la ventana de previsualización. Vamos a analizar cada una de las pestañas que lo compone.

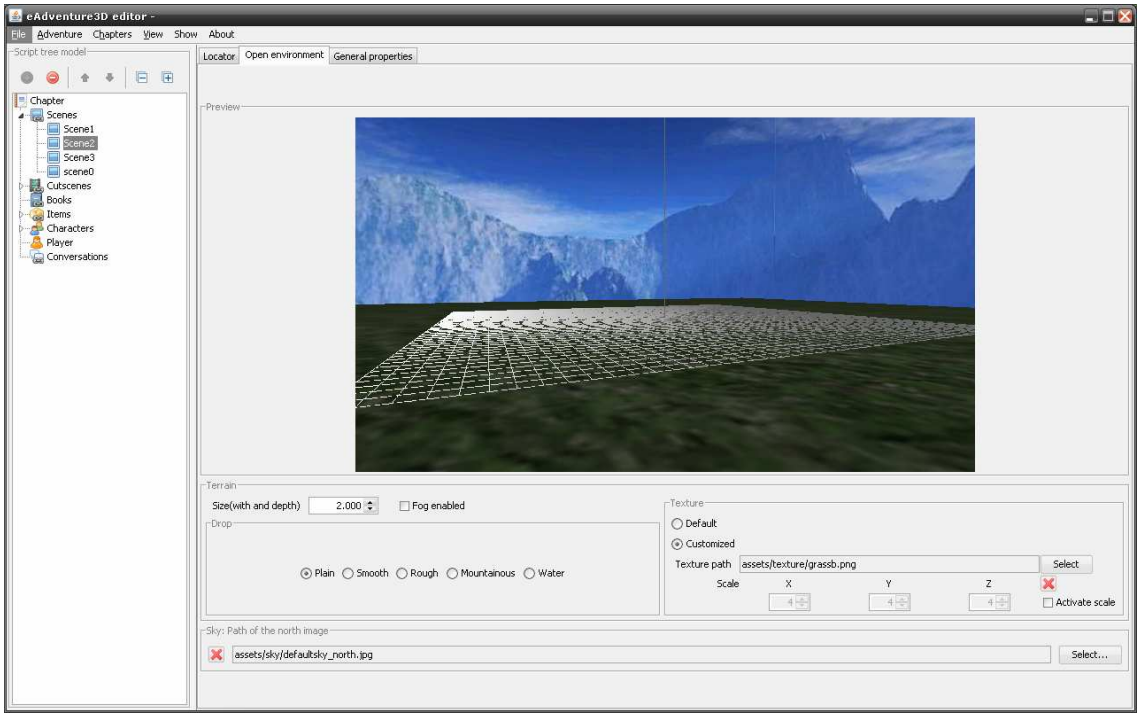
"Pestaña Locator"

Aquí nos encontramos exactamente lo mismo que lo explicado en el apartado "Pestaña Locator" de los cuartos (explicado con anterioridad) como corrobora la siguiente imagen:



"Open environment"

Lo primero que nos encontramos en "Terrain" en cuya caja podemos seleccionar la anchura del terreno, si este va a ser llano (plain), liso (smooth), rugoso (rough), montañoso (Mountainous), o en vez de terreno, agua (water). Además se puede elegir la textura del terreno, eligiendo la ruta en donde tenemos la textura correspondiente, y aplicándola una escala si fuera necesario. Por ultimo podemos elegir las imágenes de fondo (sky box), es decir, el entorno abierto, a parte del terreno, consta de 6 imágenes, una por cada lado del cubo que delimita la escena. Estas imágenes nos dan la sensación de que nos rodea, por ejemplo en las que tiene el editor por defecto, de estar rodeados de unas montañas.



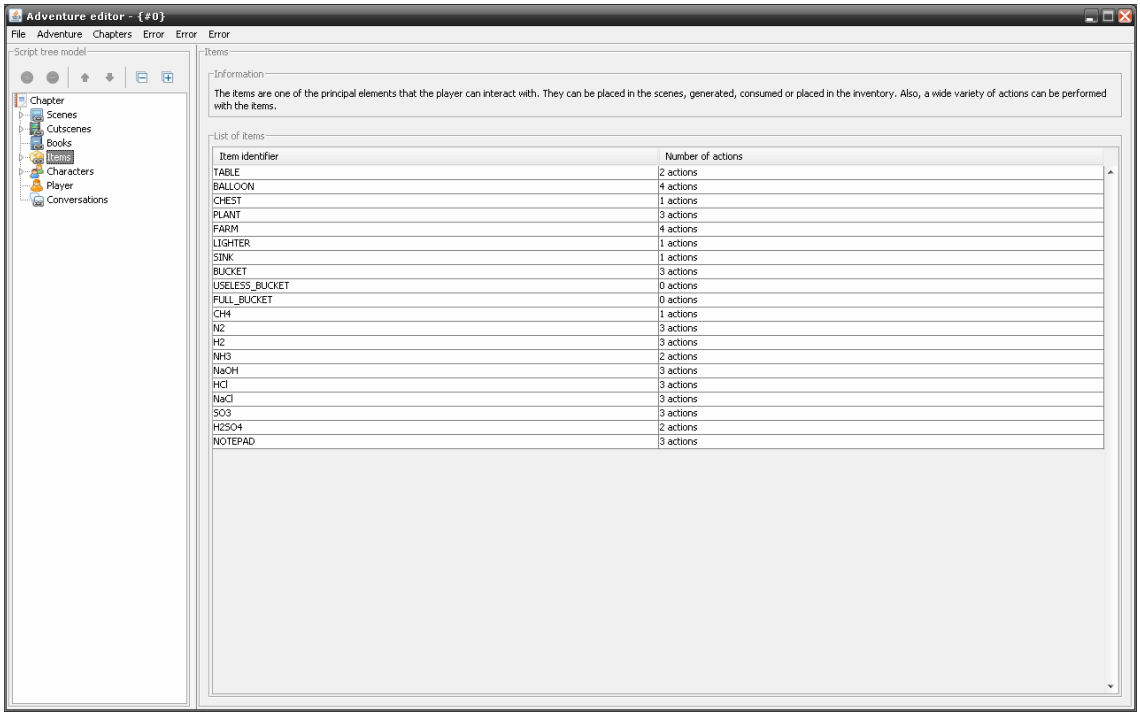
"Pestaña General properties"

En esta pestaña es igual que la de cuartos.

Cabe destacar que en entornos abiertos no tenemos forma de poner salida como vimos en los cuartos, por lo que en alguna parte de esta escena deberemos lanzar un efectos de cambio de escena o de capítulo.

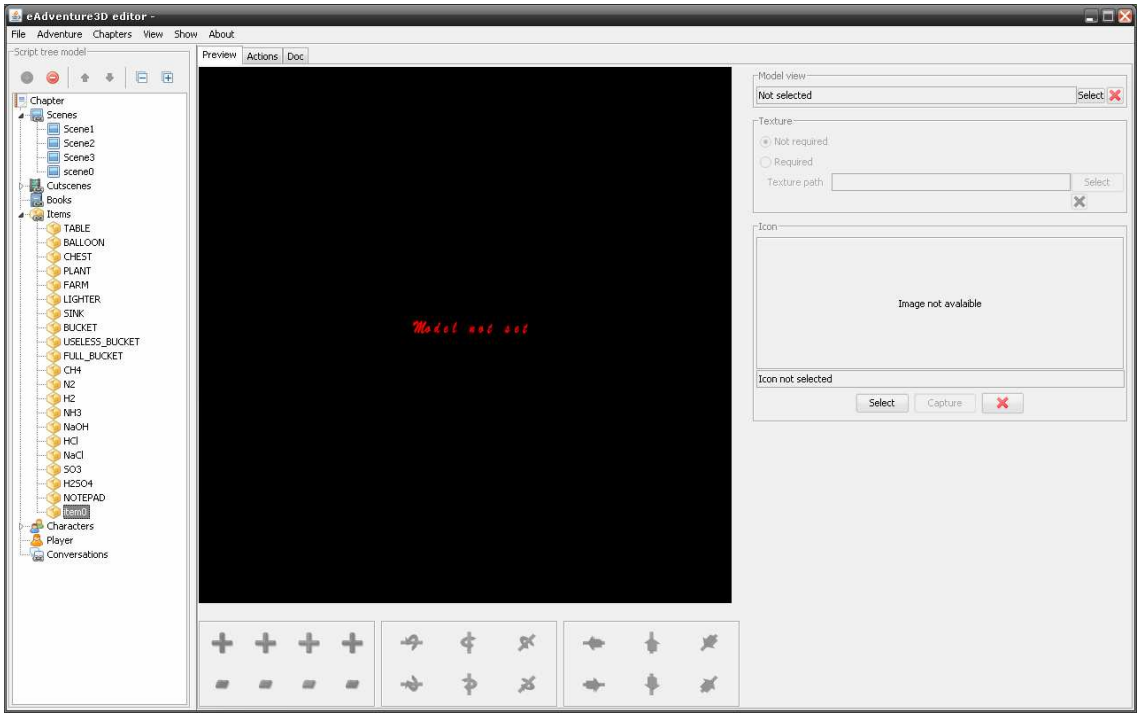
2.3. Objetos

En los juegos <e-Adventure3D> puedes definir objetos para que el jugador interactúe con ellos. Estos objetos son llamados "Ítems". Pinchando en el nodo "ítems" de la parte izquierda del panel, podemos ver todos los objetos que hemos añadido al juego, con su identificador y el número de acciones que tienen asociados. Es la misma lista que saldrá cuando queramos añadir un objeto a una escena.



Creando un nuevo objeto

Crear un nuevo objeto es más simple que crear una escena. Lo primero de todo debes hacer un clic derecho en el nodo llamado “ítems” de la parte izquierda del panel. Una vez pulsado, pincha en “Add item”. Si queremos darle otro nombre distinto al que vienen por defecto, o borrarlo, no tenemos mas que pinchar el botón derecho sobre el objeto y elegir la opción correspondiente. El editor se encargará de cambiar automáticamente todas las referencias que tenga el objeto en el juego, si es que los tuviera.



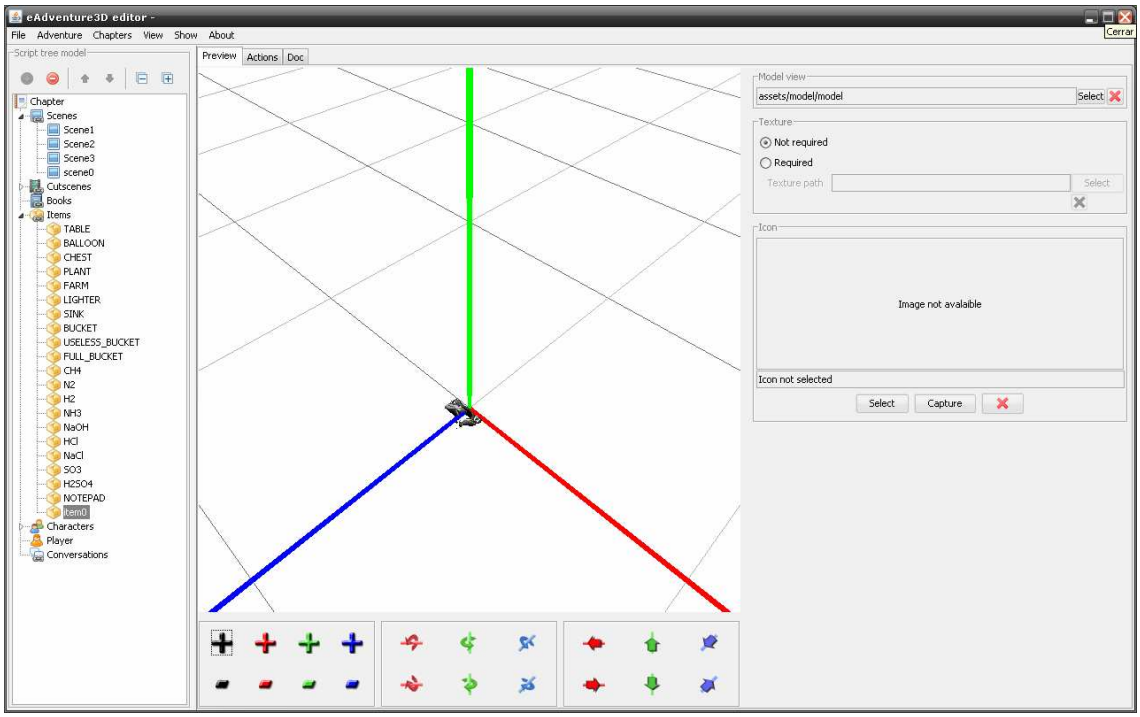
Nos aparecerá seleccionada la pestaña "Preview" con la ventana de previsualización vacía. Ahora es el momento de introducir la ruta en donde se encuentra al modelo del objeto que queremos añadir a nuestro juego <e-Adventure3D>. Para ello, debemos hacer clic en el botón "Select" del recuadro "Model view" (soporta los modelos con extensión ".3DS", ".ASE", ".DAE", ".JME", ".MD2", ".MD3", ".MS3D", ".OBJ"). Justo debajo, en el recuadro "Texture" tenemos la posibilidad de añadirle una textura al modelo, si así lo requiere. Tendríamos que seleccionar la opción "Requiered" y dar la ruta de la textura.

Llegado este punto es muy posible que el modelo que nos encontremos en la ventana de previsualización no tenga ni el tamaño ni la orientación deseada. Y aunque hasta que no lo metamos en la escena, no sabremos con precisión que valor deben tener dichos parámetros, nos conviene introducirlo para ir familiarizándonos con ello, y así poder observar todos los detalles del objeto antes de decidirnos por incluirlo en el juego. Abordamos este aspecto en el siguiente apartado.

Escalado, rotación y desplazamiento de un elemento

Vamos a poner como ejemplo que estamos añadiendo el modelo de una scooter. Observamos en la imagen que nos sale muy pequeña y torcida. Aunque el editor

por si sólo se encarga de rescalarla para que sea visible y de situar el modelo centrado en los ejes de coordenadas.



Bien, empezamos por darle un tamaño mayor. Para ellos usamos las flechas de escalado, que son los iconos "+" que encontramos abajo a la derecha de la ventana. El icono  nos va a aumentar el tamaño proporcionalmente a los tres ejes (escala el objeto). Respectivamente, disminuye. Los iconos  hacen crecer el objeto en dirección del eje de la ventana de previsualización con el que comparten color (respectivamente,  disminuyen).

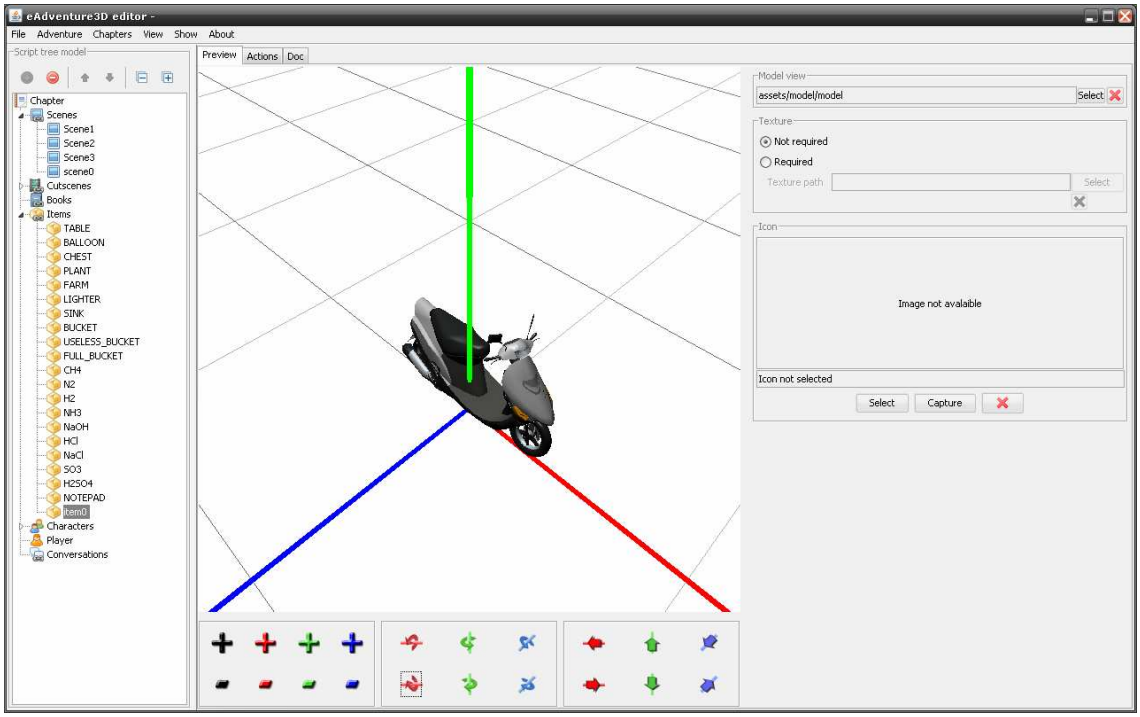
Ya tenemos el objeto a un tamaño en el que podemos apreciar los detalles. Vamos a poner la moto de pie con respecto al plano XZ. Usamos los iconos que están a la derecha de los anteriores.

El icono  nos permite girar en la dirección que indica la flecha en el eje del color que sea el icono (en este caso gira en el eje rojo, pero igual para el resto de colores). Cada clic en un icono de este tipo gira un pequeño ángulo en la dirección indicada (para ser exactos, un ángulo de 5,625º, cada 16 clics gira 90 º).

También podemos cambiar la posición y orientación de la cámara como lo

hemos hecho hasta ahora en la ventana de previsualización, moviendo el ratón mientras tenemos pulsado el botón izquierdo o movemos la rueda del ratón.

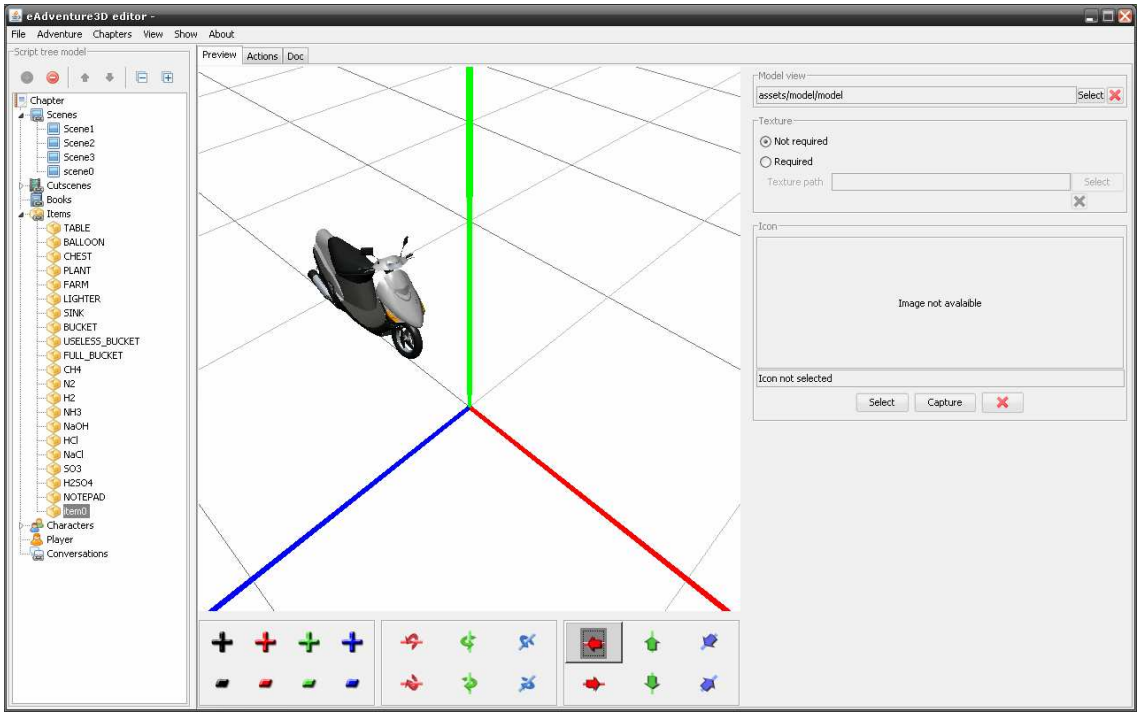
Vemos como queda:



Vamos a aprender también como desplazarla, que aunque ahora para añadir el objeto no nos va a servir mucho, si nos será útil cuando añadamos el objeto a la escena.

Usamos  para desplazar en la dirección que indica la flecha, en el eje con el que comparte color (análogo para flechas y colores distintos).

Veamos como queda tras usar el icono de ejemplo anterior varias veces:



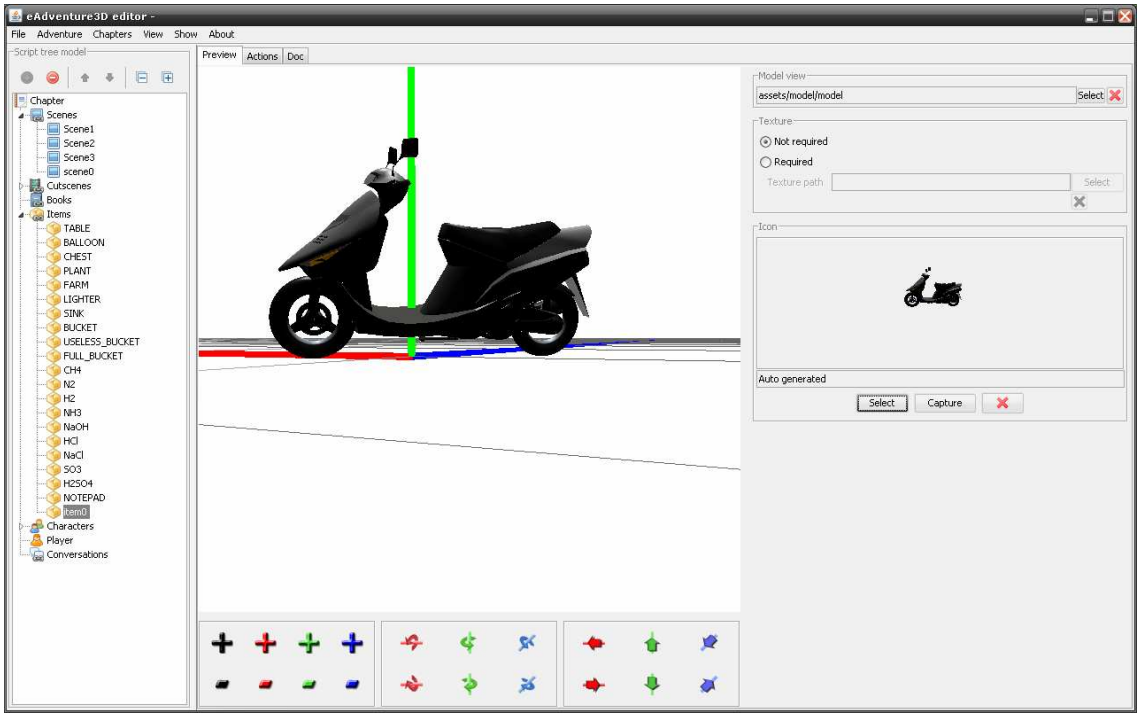
Un pequeño truco para acelerar el proceso es pulsar estos iconos de los que hemos hablado con el botón derecho del ratón. En el caso de los iconos para rotar, si pulsamos con el botón derecho el modelo girará automáticamente 90º. Para el resto de iconos simplemente se trasladará o se escalará el modelo en pasos más grandes. También otro truco es mantener pulsado el botón izquierdo en lugar de hacer varios clics. En este caso, los resultados irán aumentando proporcionalmente al tiempo que pulsemos.

Cuando posteriormente lo añadamos a una escena, nos saldrá a igual distancia de los ejes, y con la posición y tamaño decididos aquí, pudiendo ser refinada cualquiera de los parámetros estudiados en este apartado, como veremos más adelante al hablar de incluir elementos en una escena.

Haciendo un icono para el inventario.

Es posible que decidamos que el jugador puede añadir este objeto a su inventario durante el desarrollo del juego. El editor nos da la posibilidad de crear un icono para representar dicho objeto en el inventario. No tenemos más que orientar la cámara de la ventana de previsaulización y pensar que el icono va a quedar como lo que estamos viendo en este momento en la ventana.

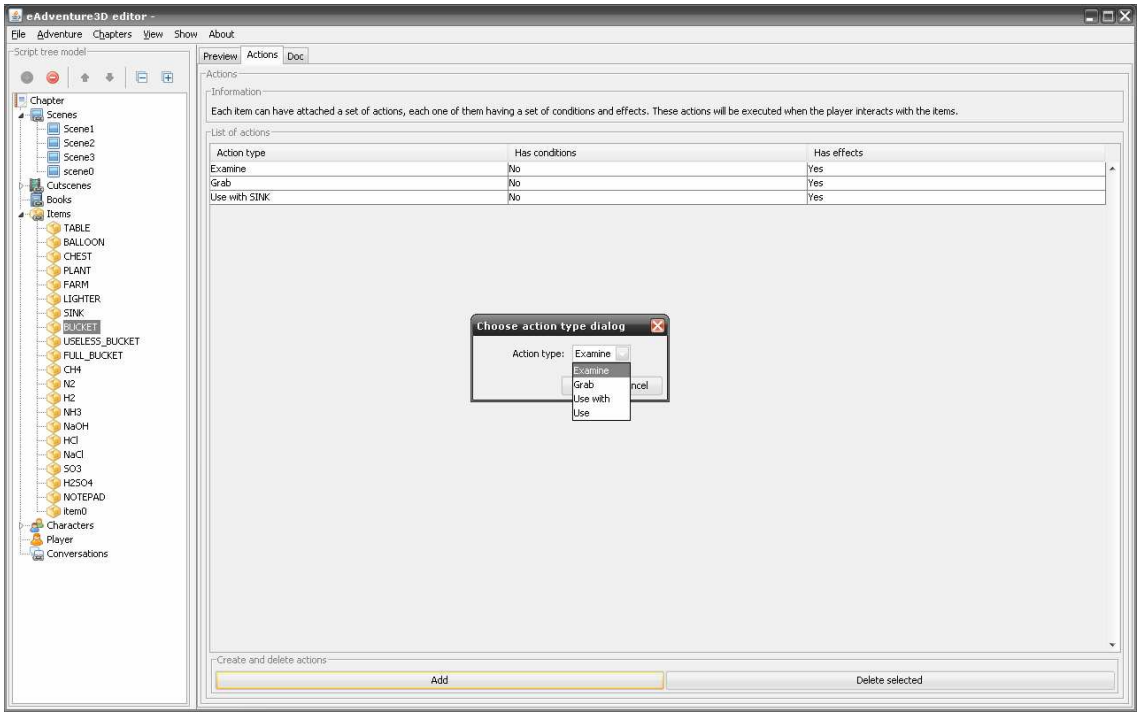
Una vez elegida como va a quedar el icono, pulsamos en “Capture icon for model preview” en el recuadro “Icon”, a la derecha de la ventana. Nos saldrá la previsualización del icono como se ve el la siguiente imagen.



Si queremos crear nuestro propio icono, también podemos añadir una imagen cualquiera para que sea la representación del objeto en el inventario. Simplemente tenemos que pinchar en el botón “Select” y nos aparecerá una ventana para seleccionar la ruta de la imagen en cuestión.

Interactuando con los objetos: Acciones

Ahora nuestro objeto esta satisfactoriamente creado. Pero darle la apariencia deseada a nuestro objeto no es lo único que podemos hacer. Al lado de la pestaña “Preview” no encontramos la pestaña “Actions”. Nada mas elegir esta pestaña, nos aparecen las acciones que hemos asociado anteriormente al objeto. Al pulsar sobre el botón “Add” del recuadro “Create an delete acctions” de la parte inferior nos aparece una ventana con las interacciones que se pueden especificar. Son las siguientes:



Examinar ("Examine")

Cuidado de no confundir examinar con mirar. Todos los items en <e-Adventure3D> pueden ser examinados. Cuando miras un objeto en el juego, aparece en pantalla una breve descripción del mismo, como explicaremos mas adelante. Por el contrario, cuando un objeto es examinado, se muestra una descripción detallada del mismo. No puedes cambiar el comportamiento de mirar un objeto, pero la acción de examinar sí es configurable. Puedes usar estos dos tipos de comportamiento para obtener primero una visión general, y después una descripción mas extensa.

Coger ("Grab")

Cuando añades la acción coger a un objeto, permites que este sea cogido por el jugador, desapareciendo de la escena, y siendo añadido al inventario. Entonces el jugador podrá usarlo donde quiera usarlo.

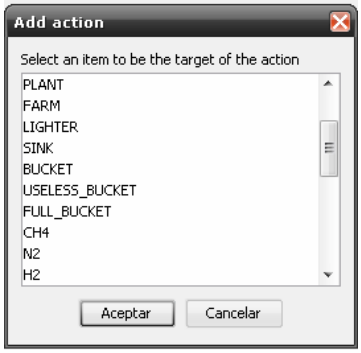
Usar ("Use")

El objeto podrá ser usado por sí mismo. Esto significa que podrás usarlo y

lanzar los efectos que tenga asignados.

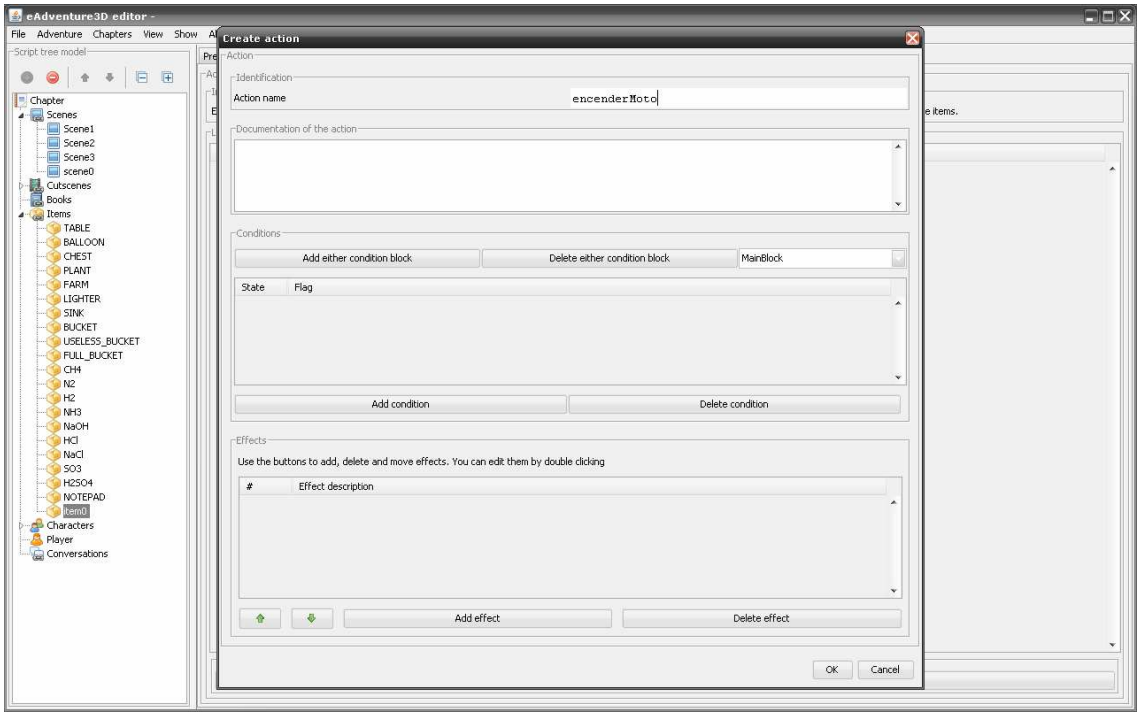
Usar con ("Use with")

Es equivalente a usar, pero en este caso el objeto no se podrá utilizar por sí solo, sino que necesitarás usarlo con otro objeto o personaje para producir los efectos asociados a dicha acción. Nos aparecerá una lista con todos los objetos y personajes existentes en el juego, para elegir con cual debemos usarlo para que se produzca el bloque de efectos deseado. Para 'usar con' el objeto debe estar en el inventario, esto debe ser tenido en cuenta.



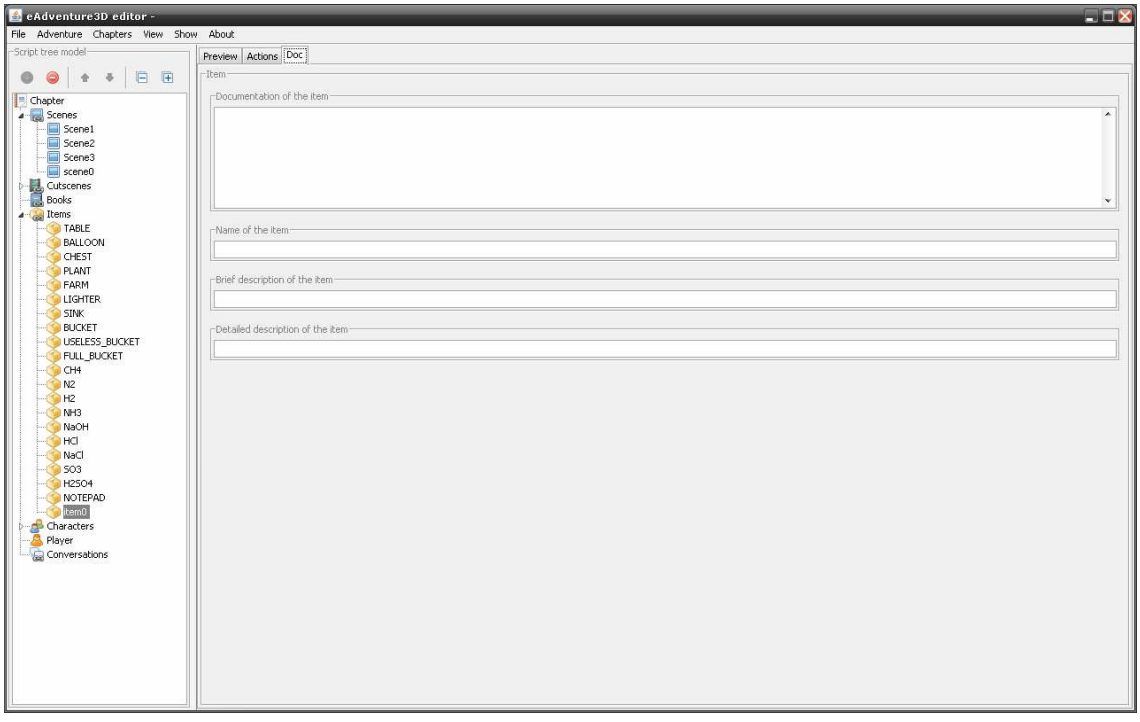
Para todas las acciones

Una vez seleccionado el tipo de acción, nos aparece la ventana de edición de las propiedades. Aquí podemos darle nombre a la acción creada, añadirle cualquier tipo de información complementaria y editar las condiciones y efectos. Esto último será analizado en la sección 3.



Volveremos a las acciones cuando hablemos de acciones y efectos. Esto ha sido solamente una introducción sobre lo que podemos hacer y lo que no con objetos.

Documentación

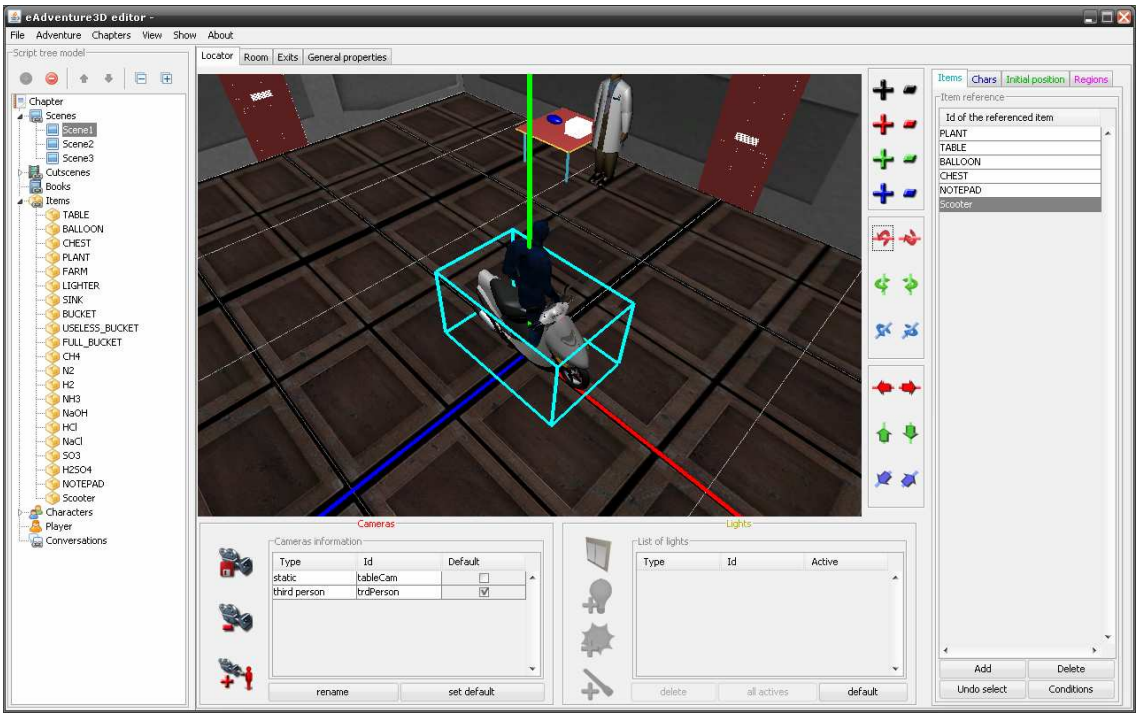


Como para las escenas, podemos añadir información para los objetos. Lo haremos en la pestaña que está a la izquierda de “Actions”, pinchando en “”Doc”. Daremos un breve vistazo a esto. Como puedes ver, hay cuatro recuadros que podemos rellenar. El primero nos permite dar una completa especificación de lo que el objeto es. Los otros tres serán usados para mostrar información sobre el objeto en el juego. El nombre (“Name of the item”) es el texto que aparecerá en el “hud” cuando nos acerquemos al objeto en el juego. El recuadro “ Brief description of the item” será una descripción corta del objeto, y el recuadro “Detailed description of the item” será rellenado con lo que aparecerá cuando examinemos un objeto.

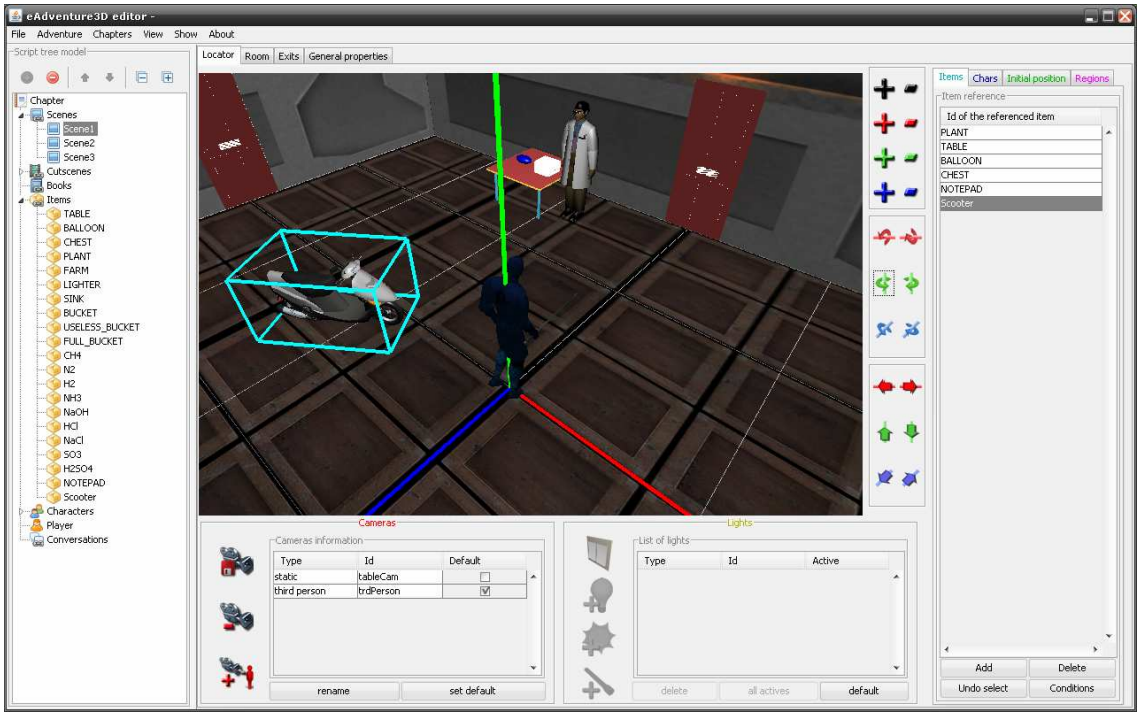
Añadir un objeto a la escena

Una vez un objeto ha sido creado, podemos referirnos a él en cualquier parte del capítulo. Por ejemplo, podemos crear un objeto que esté presente en varias escenas. Ahora aprenderemos como añadir un objeto en las escenas.

El primer paso es seleccionar el nodo de la escena en el árbol de la derecha. En la pestaña “Locator” encontramos a su derecha unas pestañas, de las cuales debemos seleccionar la de objetos (“Items”). Una vez aquí, pulsamos en el botón “Add” de la parte inferior. Nos saldrá una lista con todos los objetos existentes en el juego, eligiendo el que queramos añadir. En nuestro caso vamos a añadir la scooter que acabamos de crear. Lo elegimos, y nos aparecerá el objeto en la posición relativa a los ejes, con el tamaño y posición de como fue creado:



Ahora vemos que donde la colocamos al crearla no queda bien. No tenemos mas seleccionarla y usar los iconos de escalado, rotación y desplazamiento que aparecen a la izquierda de la ventana de previsualización, de igual manera que vimos anteriormente.



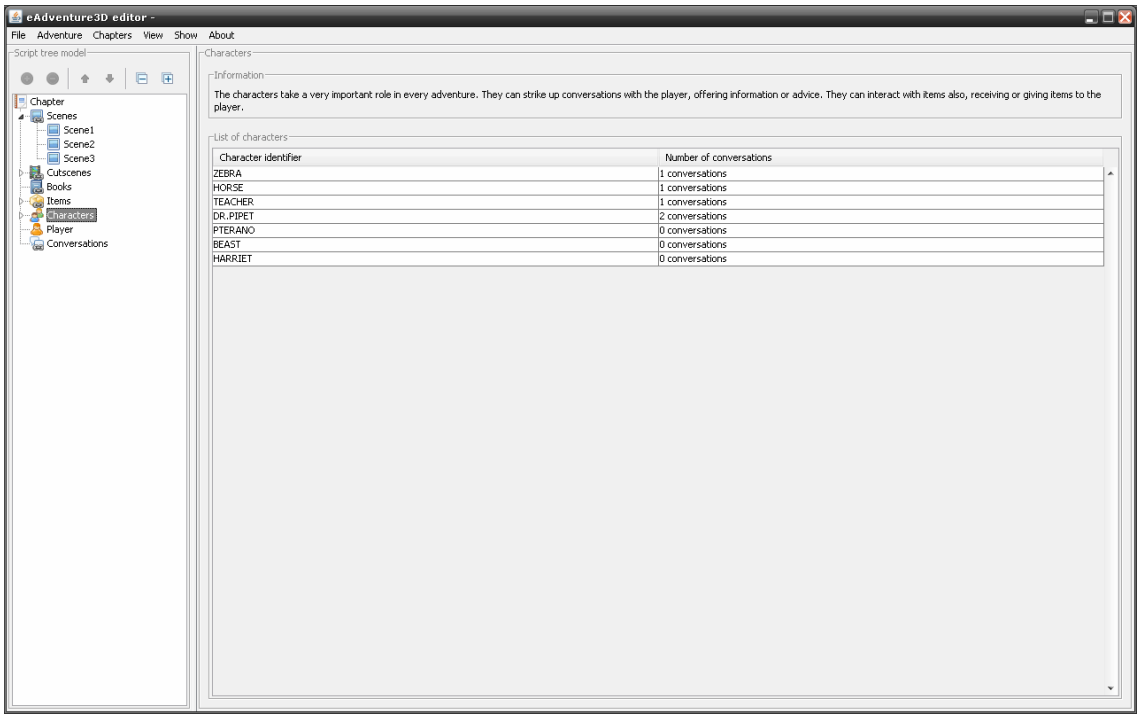
2.4. Personajes

Un personaje es un elemento con el cual el jugador puede hablar. En esta sección vamos a aprender a crear y editar los aspectos relacionados con los personajes.

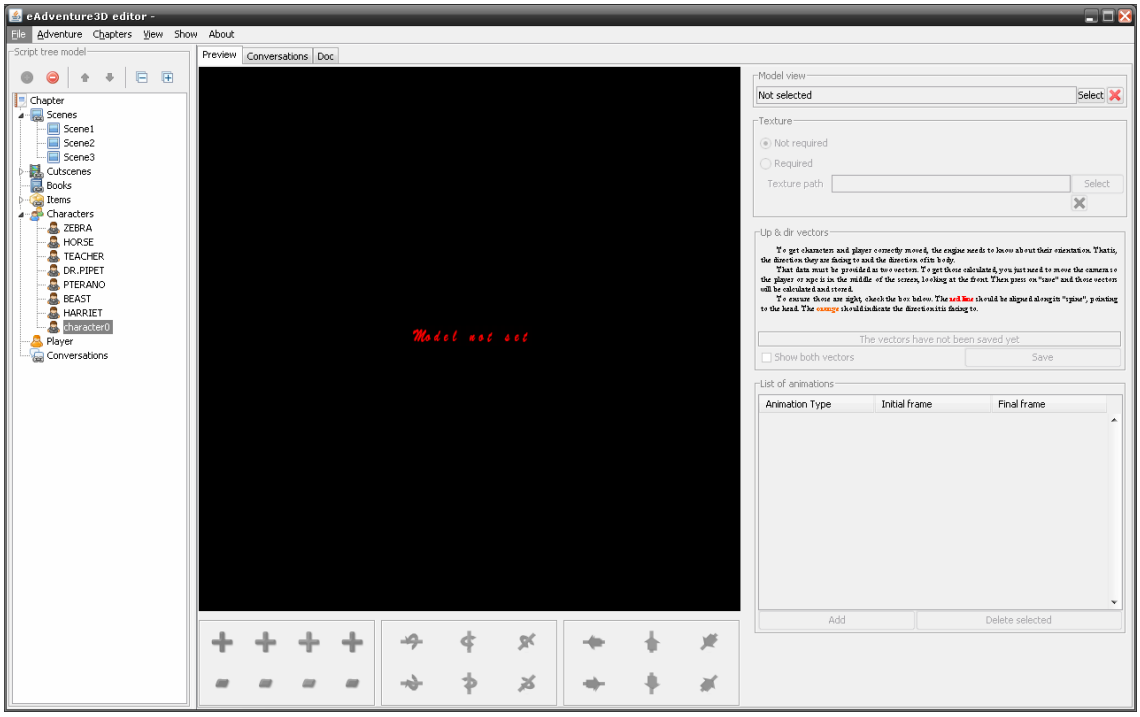
Creando un nuevo personaje

Añadir nuevos personajes al capitulo sigue las mismas reglas simples que los objetos. Para este propósito hay un nodo en el árbol etiquetado como “Characters”.

Pinchando sobre el nodo nos aparece la lista de personajes existentes hasta el momento con el número de conversaciones que tiene cada uno.



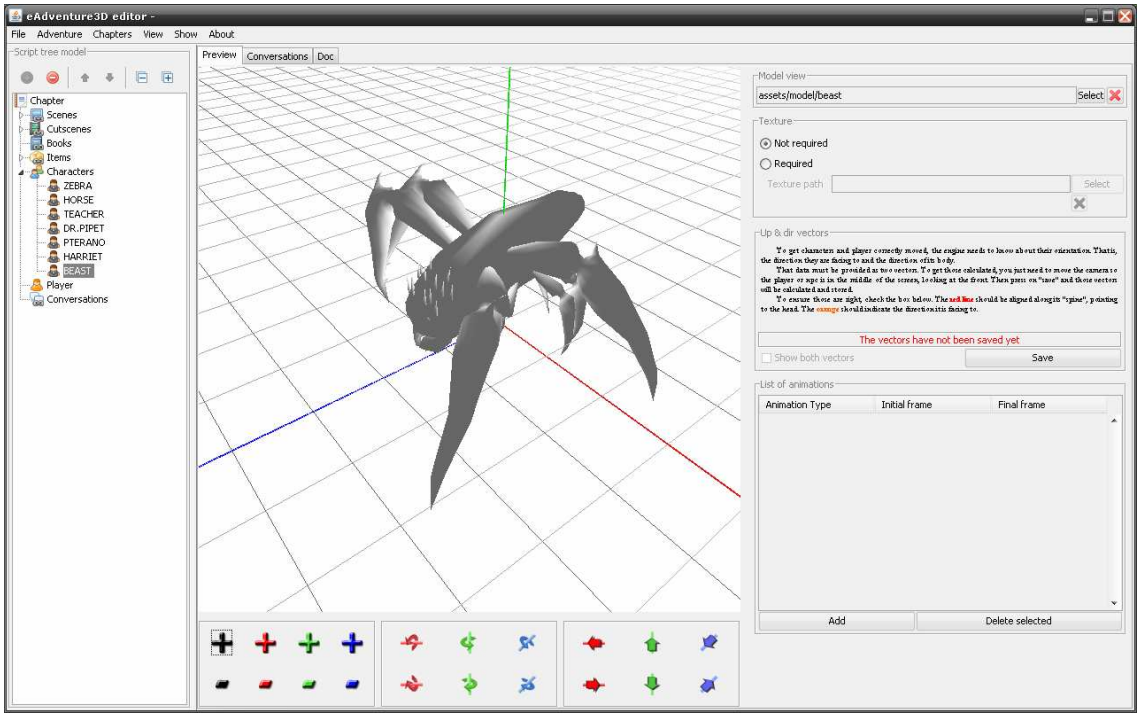
Pinchando con el botón derecho sobre este mismo nodo, nos aparece la opción "Add NPC" (NPC – "non player character", o lo que es lo mismo, personaje no principal). Eligiéndola nos aparece la siguiente pestaña ("Preview"), muy parecida a la que nos salida cuando creamos un nuevo objeto:



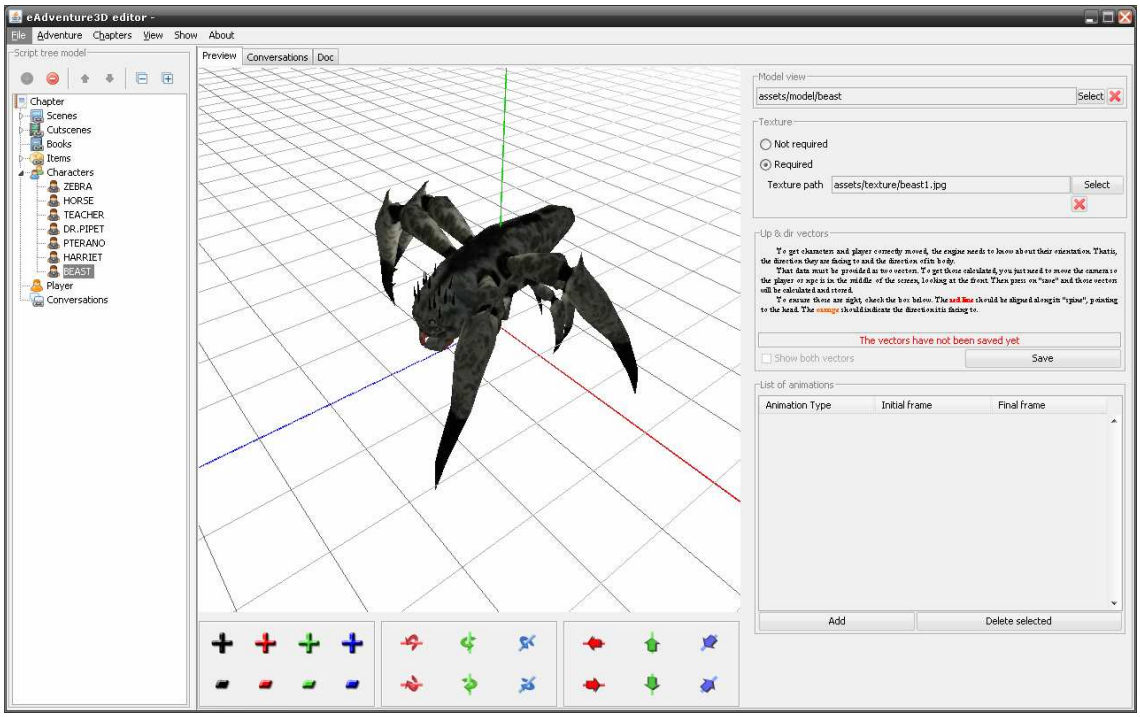
Si queremos cambiarle el identificador, clikeamos con el botón derecho sobre el nodo nuevo que se ha creado en el nodo “Characters” del árbol principal, y seleccionamos “Rename element”, o “Delete element” si lo queremos eliminar.

De la misma manera que en objetos, lo primero que debemos hacer es cargar el modelo del personaje que vamos a incluir. Para ello elegimos la ruta donde se encuentre dicho modelo en el recuadro “Model view” que se encuentra en la parte superior derecha. Si necesita textura, seleccionamos la ruta en el recuadro “Textura” que se encuentra inmediatamente debajo, seleccionando la opción “Requiered”.

Vemos como este modelo necesita textura:



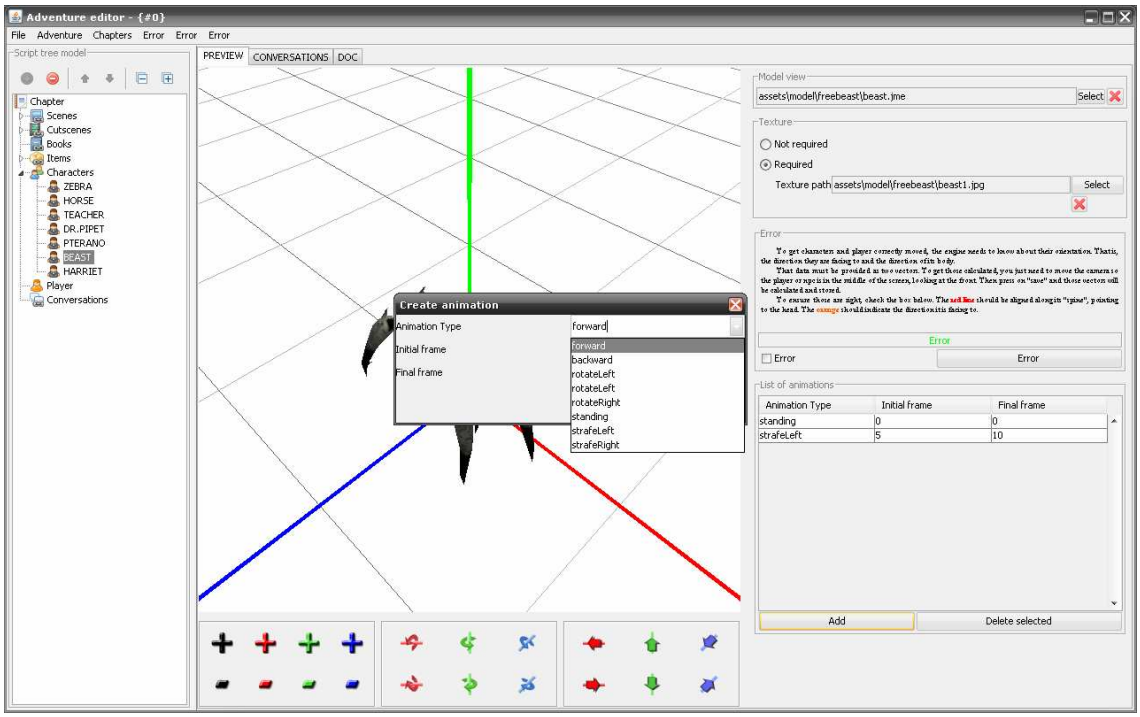
Y como queda al añadirla:



Ahora tenemos que fijar las animaciones que queremos utilizar dentro de las posibilidades que nos de cada modelo. Podemos asignar varias configuraciones de animación para diversas situaciones de juego (con el modelos suele venir información relativa a los frames que lleva asociado cada movimiento del modelo).

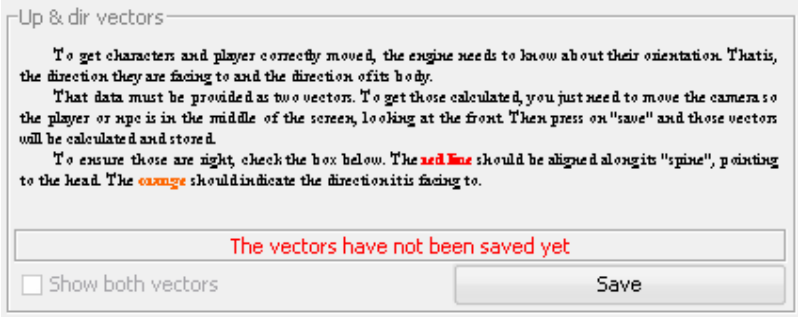
Veamos como añadir una nueva:

Pulsamos en el botón “Add” del recuadro “List of animations” que se encuentra en la parte inferior derecha. Nos sale una lista para que elijamos el tipo de animación que es (hacia delante, hacia atrás, rotar izquierda, rotar derecha, permanecer quieto, strafe izquierda y strafe derecha, en ese orden). Esta lista es editable por si se quiere añadir una animación con otro nombre distinto y se pueda lanzar luego a través de un efecto. Debajo nos pide el frame inicial y final para esa animación en concreto. Pulsamos en “Ok” y tendremos la nueva animación definida para este personaje.



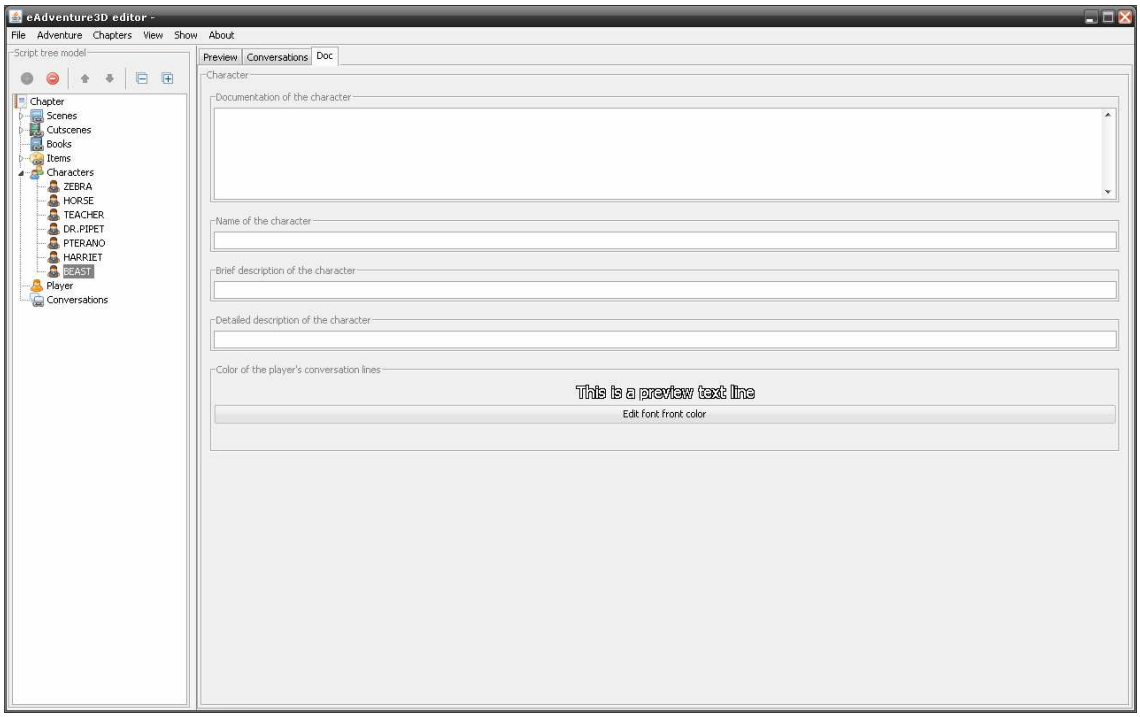
Para que los personajes y jugadores que utilizan modelos animados puedan ser correctamente trasladados, el motor necesita saber su orientación. Esto es, la dirección a la que miran y la dirección vertical de su cuerpo. Si seleccionamos “Show both vectors”, nos serán mostrados estos ejes en el modelo. Estos ejes serán alineados según esté mirando el modelo y a lo largo de la columna vertebral señalando hacia la cabeza respectivamente.

Para guardar correctamente estos vectores hay que orientar el modelo de tal forma que mire hacia fuera de la pantalla y permanezca con la cabeza hacia arriba, para lo cual es útil el menú superior “View”, y seleccionamos el botón “Save”:



Documentación

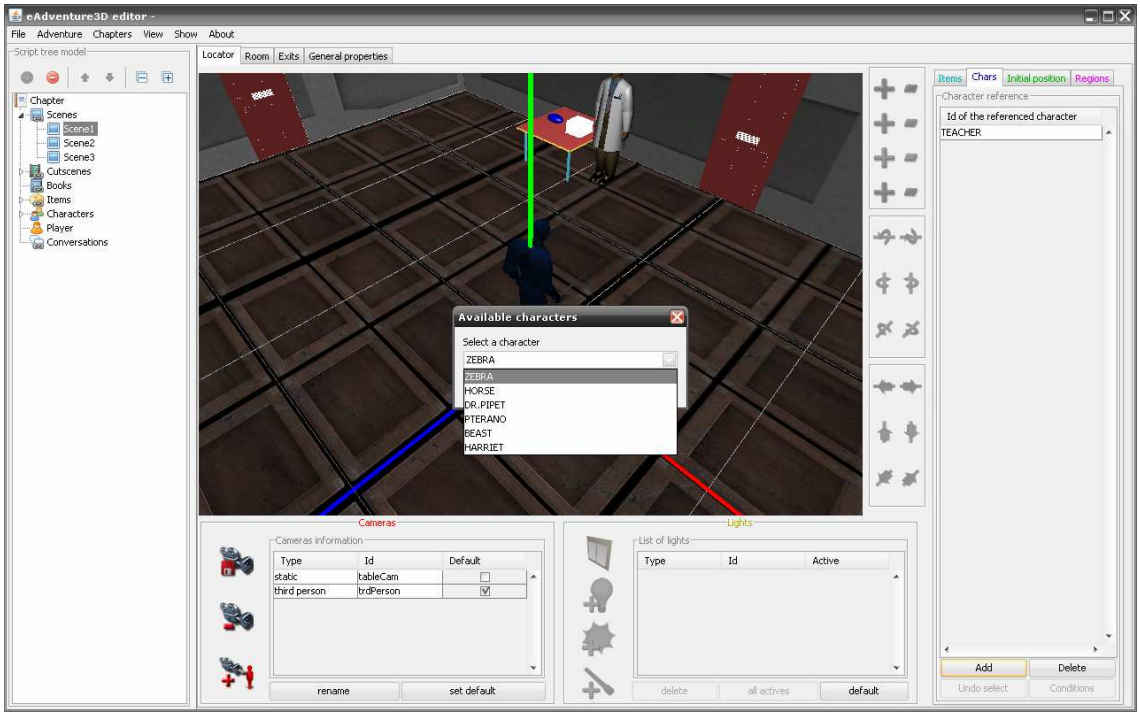
Como en los objetos, a los personajes se les puede adjuntar documentación. Veamos que hay un campo más aquí respecto de la pestaña "Doc" de los objetos. Para los personajes se puede editar el color con el que aparecerán los comentarios de dicho personaje en una conversación en el juego. Esto es especialmente interesante cuando más de un personaje interviene en una conversación subtitulada. El resto de campos tienen la misma aplicación que para los objetos.



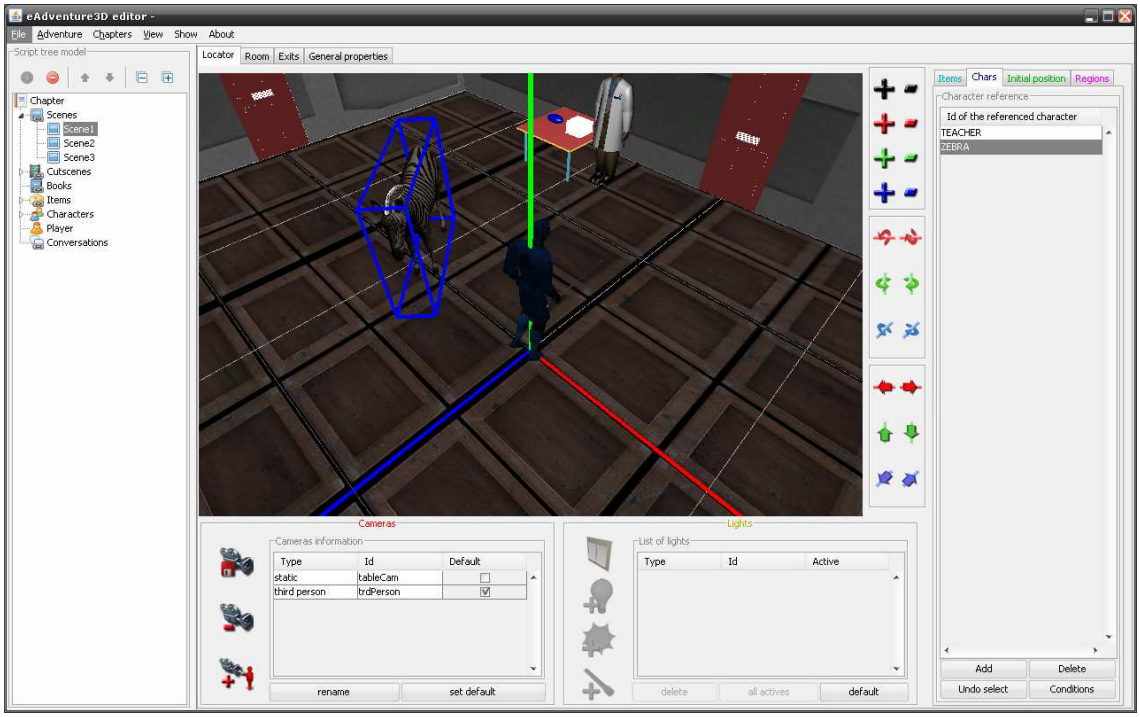
Añadir personajes a una escena

De igual manera que para objetos, podemos referenciar personajes en las escenas. Es entonces cuando un personaje aparecerá en la escena, pudiendo interactuar con el jugador (o personaje principal). Para conseguir esto solo

necesitamos selecionar el nodo de escena donde lo queremos añadir, seleccionar la pestaña “Chars” de las cuatro pestañas que encontramos a la derecha de la ventana de previsualización (dentro de la pestaña “Locator”).



Pinchamos sobre el botón “Add” de la parte inferior, y nos sale una lista con los personajes creados para este capítulo. Elegimos uno, y lo situamos en donde queramos de igual manera que lo hicimos con los objetos (con los iconos de rotación, escalado y desplazamiento).



Conversaciones

Igual que se puede interactuar con los objetos, también podemos hacerlo con personajes. Pero esta interacción es significativamente diferente. Con los personajes no podemos especificar acciones. A diferencia si podemos especificar conversaciones, las cuales son sucesiones de líneas de texto que son habladas tanto por el jugador principal como por personajes. Son muy útiles por dos motivos: guiar al jugador en el juego y proveer en algunas ocasiones como fuente de información.

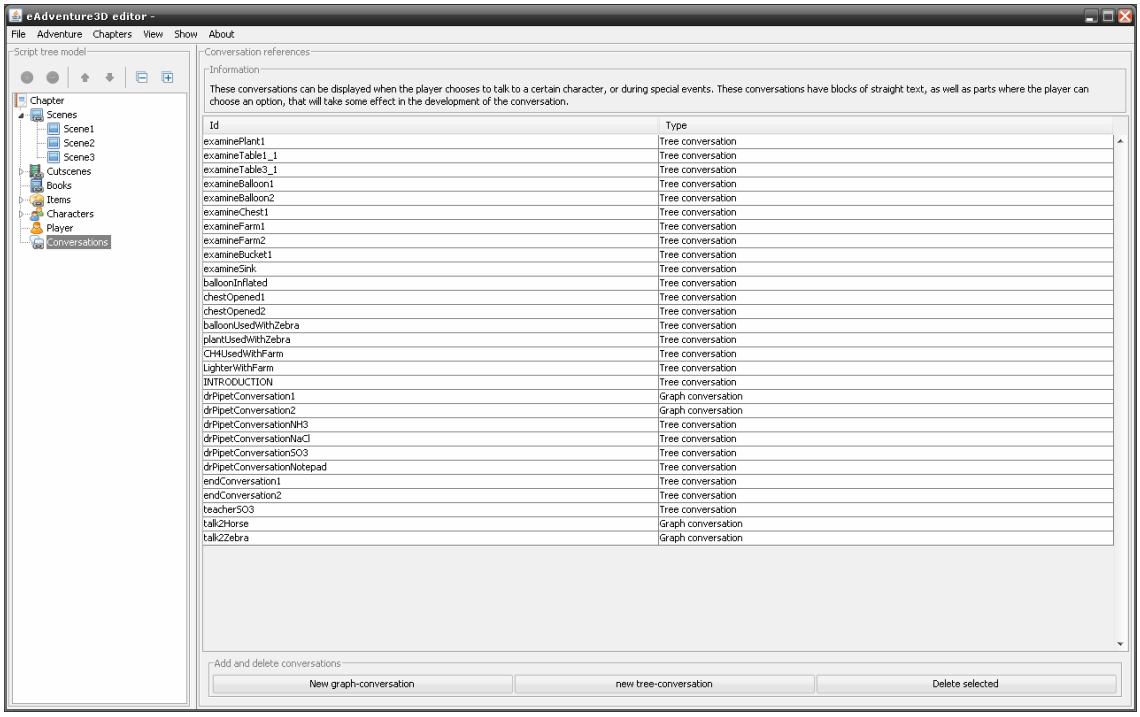
Más de un personaje y el jugador pueden tomar parte en una conversación. Como en los juegos convencionales de Lucas Arts el jugador puede en algunos casos elegir cual va a ser la siguiente línea que va a decir de una lista de opciones. Dependiendo de la opción elegida la conversación tomará una rama u otra.

Las conversaciones se adjuntan a los personajes, por lo que serán lanzadas cuado el jugador se acerque a un personaje y elija la opción de hablar con él/ella. Para crear una nueva conversación y adjuntarla a un personaje en concreto, debemos acudir al nodo “Conversations” del árbol. Una vez que una conversación ha sido creada, podemos adjuntarla a un personaje.

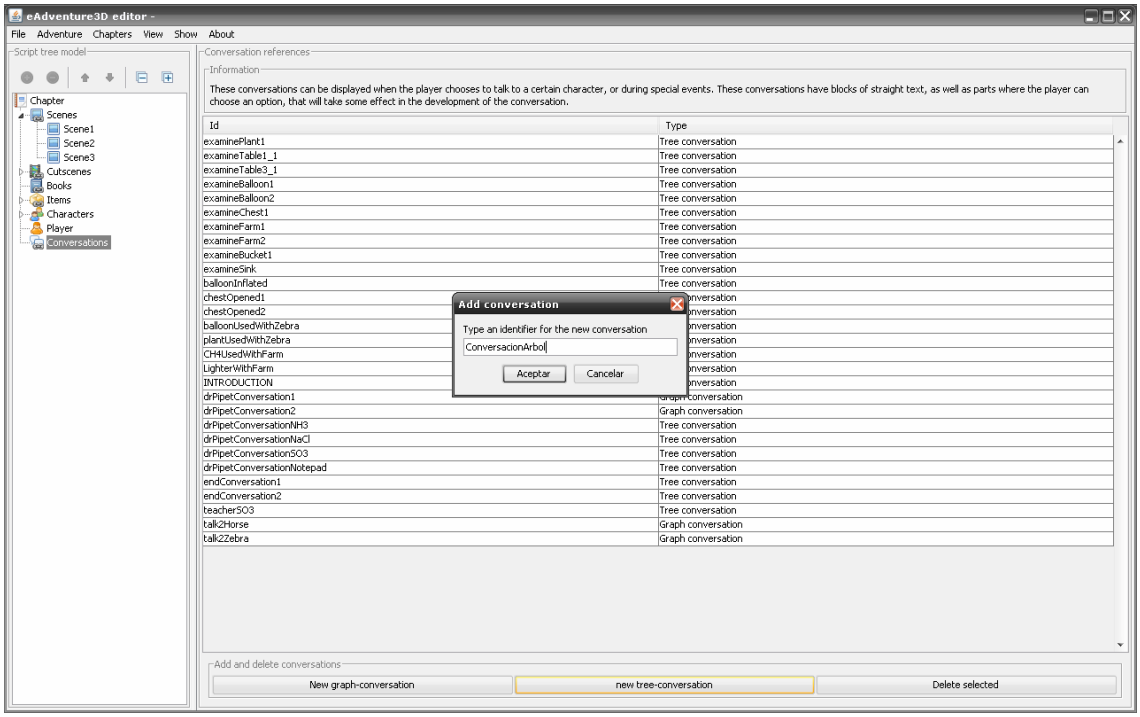
Ahora vamos a ver como crear conversaciones. Primero debes saber que hay dos tipos de conversaciones.

Conversaciones de tipo árbol (Tree conversations)

Una conversación de tipo árbol apropiada cuando no hay que soportar bucles en la conversación. En este tipo de conversaciones siempre se llega al final. Si hacemos clic en el nodo “Conversations” nos aparecen todas las conversaciones que hemos creado para este capítulo.



Para crear una nueva conversación de tipo árbol debemos pinchar en el botón “New tree conversation”, pidiéndonos el identificador para esta conversación.



Una vez introducido, nos sale el editor de conversaciones. Una conversación será dibujada como un conjunto de nodos (pintados en círculos negros), los cuales están unidos a otros nodos. Hay **tres tipos de nodos:**

Nodos de dialogo

Dichos nodos contienen un conjunto de líneas que son hablados por el jugador o el personaje, en el orden especificado.

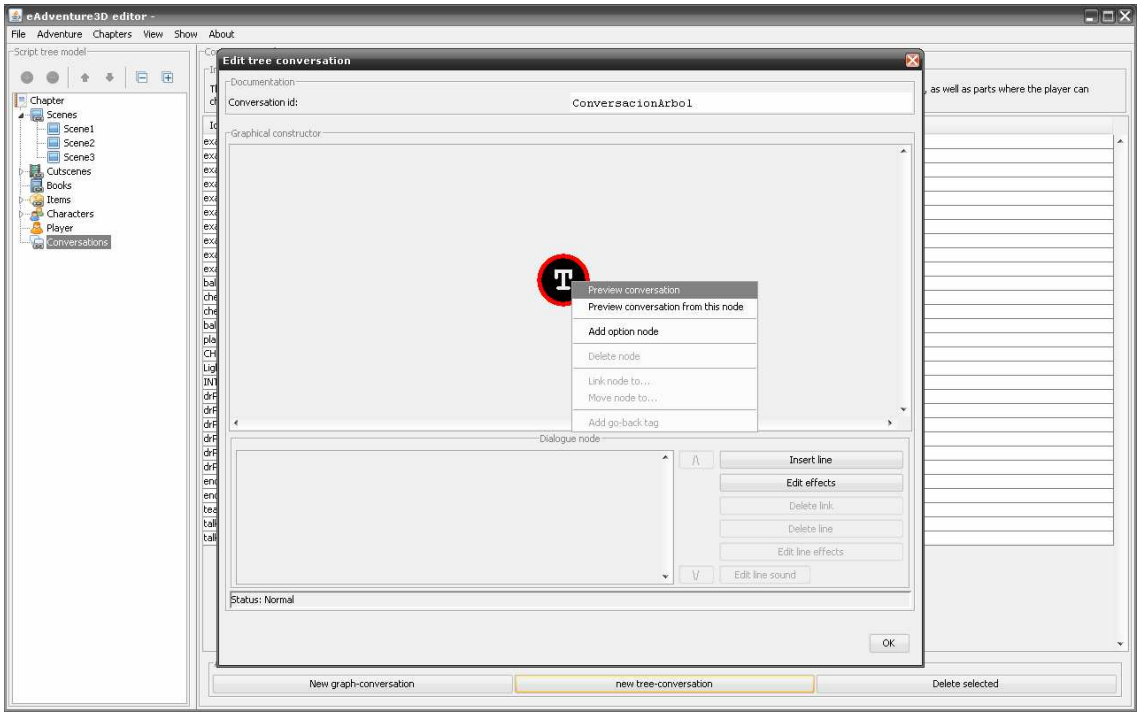
Nodos de opción

Estos nodos contienen opciones para elegir una cuando la conversación llega a uno de este tipo. La opción seleccionada será dicha por el jugador en la siguiente línea de conversación, y llevará la conversación desde este nodo de opción a otro nodo.

Nodos terminales

Estos son nodos de dialogo. La única diferencia es que este tipo de nodos no pueden ser unidos a otros, por lo que la conversación acabará cuando se llegue a un nodo de este tipo (después de que el contenido de este nodo haya sido completamente hablado).

Vamos a comenzar a editar la conversación (ventana que nos sale justo después de introducir el identificador de conversación). Podemos seleccionar cualquier nodo que tengamos creado pinchándolo con el botón izquierdo del ratón. Así se pondrá el nodo rodeado de rojo, en vez de negro, y los campos que puedes editar se mostrarán en la parte inferior derecha. Con el botón derecho del ratón se muestran las opciones para unir de diferentes maneras un nodo con otros, así como hacer una previsualización tanto de la conversación entera, como de la conversación de este nodo en adelante.

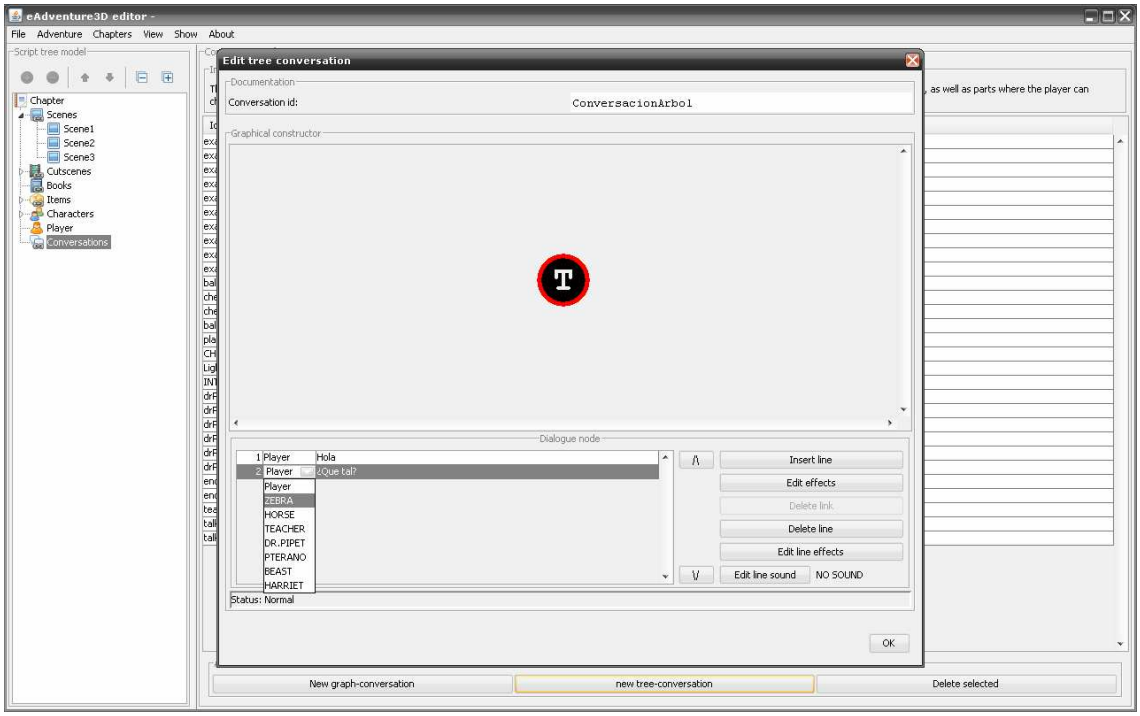


Por ejemplo, seleccionamos el nodo que tenemos cuando creamos una conversación. Nos damos cuenta de que es terminal con una gran “T” dibujada en su interior. Entonces usamos los botones “Insert line” y “Delete line” para insertar y borrar nuevas líneas de conversación que serán habladas por el jugador u otros personajes. Los botones de arriba (“up” /\ ) y abajo (“down” \/) pueden ser usados para cambiar el orden en el que cada línea de conversación debe ser dicha. Cuando una línea es seleccionada en la tabla, usaremos estos dos botones para

moverla arriba o abajo, siendo dicha en una posición previa o posterior. También podemos especificar una pista de audio que será reproducida junto con la línea de texto. Para ello usa el botón “Edit audio”. Gracias a esto podemos grabar el texto de una línea en un archivo de audio, y poder escuchar la conversación en el juego.

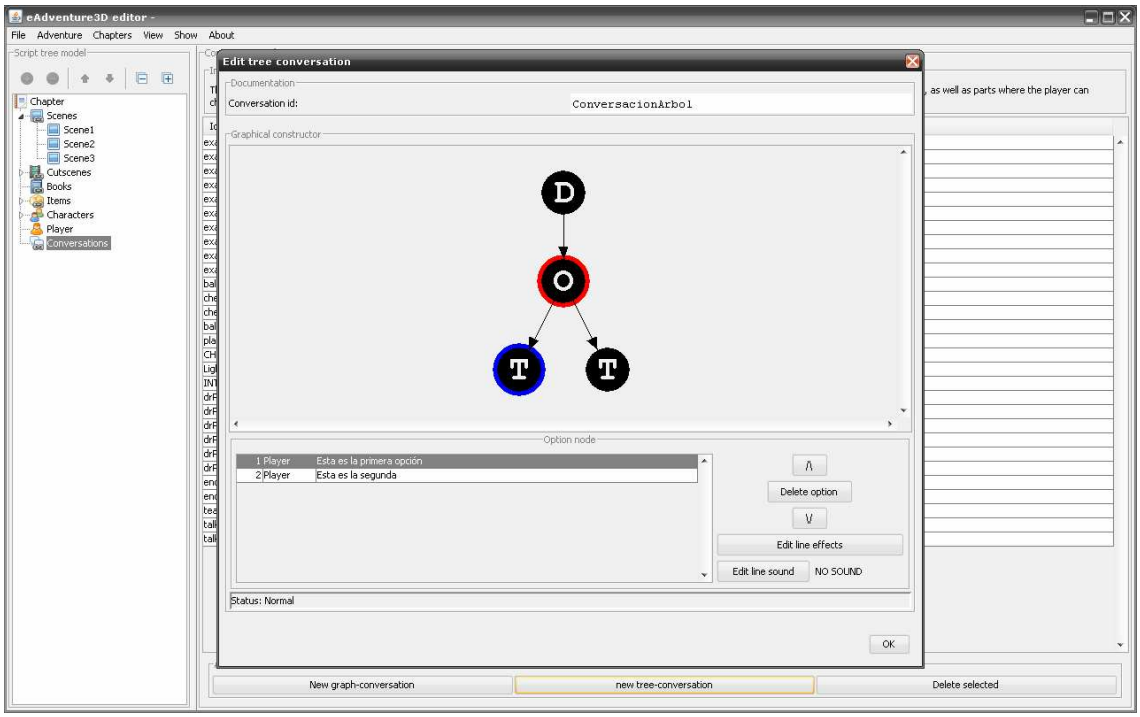
Hay dos botones más que podemos usar. El primero, etiquetado como “Editar efectos” nos permitirá especificar un bloque de efectos para ser ejecutados después de que todas las líneas del nodo hayan sido dichas, o después de cada línea. El otro, “Delete link”, se usa para borrar uniones entre nodos.

Ahora sabemos lo que necesitamos, vamos a editar una conversación. Por ejemplo, podemos insertar dos líneas. La primera será dicha por el jugador, y la otra por el personaje. Como se puede ver, para cada línea podemos cambiar quien la dirá por medio de una lista, y el tipo de texto de la línea.

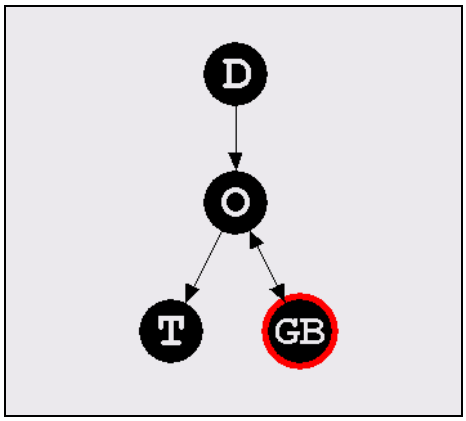


Ahora, queremos que el jugador elija entre dos opciones. Por lo tanto debemos crear un nodo de opciones y unirlo al nodo actual. Para hacer esto, clickeamos con el botón derecho sobre el nodo y seleccionamos “Add option node”. Una vez echo esto vemos que aparece un nuevo nodo con el texto “O” dibujado en su interior, colgando del primer nodo. Este no será terminal nunca mas, ya que hay otros nodos por debajo (por ello llevan una “D” dibujada en vez de la “T” inicial). Para añadir opciones en este nodo, hay que seleccionarlo, clickear con el botón derecho y seleccionar “Add dialogue node”. Veremos en el panel de la parte inferior que

una nueva línea ha sido creada a lo largo con un nuevo nodo de dialogo colgando del nodo de opciones. Si seleccionamos esta línea de opción, el nodo al cual iremos eligiendo esta opción en el juego se remarcará en azul. Nos queda rellenar el texto de la opción (el cual será dibujado en la lista de opciones y dicho por el jugador cuando sea seleccionado). Por ejemplo, nosotros escribimos “Esta es la primera opción”. Ahora creamos otra opción y escribimos “Esta es la segunda opción”. En este punto nuestra conversación tiene la siguiente apariencia:



Siguiendo este mecanismo puedes crear conversaciones de tipo árbol tan grandes como quieras. Sin embargo, en ocasiones nos gustaría añadir opciones en un nodo de opciones para volver atrás en el nodo. De esta forma, cuando esta opción sea elegida, todas las opciones serán mostradas y el jugador tendrá que elegir otra de nuevo. Esto se puede conseguir clickeando con el botón derecho en la opción que queremos que haga volver a atrás, y seleccionamos la opción “Add go-back tag”. Entonces, cuando todas las líneas de conversación en este nodo hayan sido dichas, la conversación volverá al nodo de opciones anterior.



Cuando hemos finalizado de editar la conversación, podremos previsualizarla como será en el juego. Para ello presionamos el botón derecho del ratón en cualquier nodo, y seleccionamos la opción “Preview conversation” o “Preview conversation from this node”, para ver la conversación entera, o desde el nodo seleccionado en adelante, respectivamente.

Conversaciones de tipo grafo (Graph conversations)

Además de las conversaciones de tipo árbol, podemos crear conversaciones de tipo grafo de una manera similar. La diferencia con las conversaciones de árbol es que están permitidos los bucles. Por ello podemos unir un nodo con cualquier otro. Para hacer este proceso más sencillo, podemos mover los nodos a lo largo del panel, para que no se solapen unos a otros.

Añadiendo una conversación a un personaje

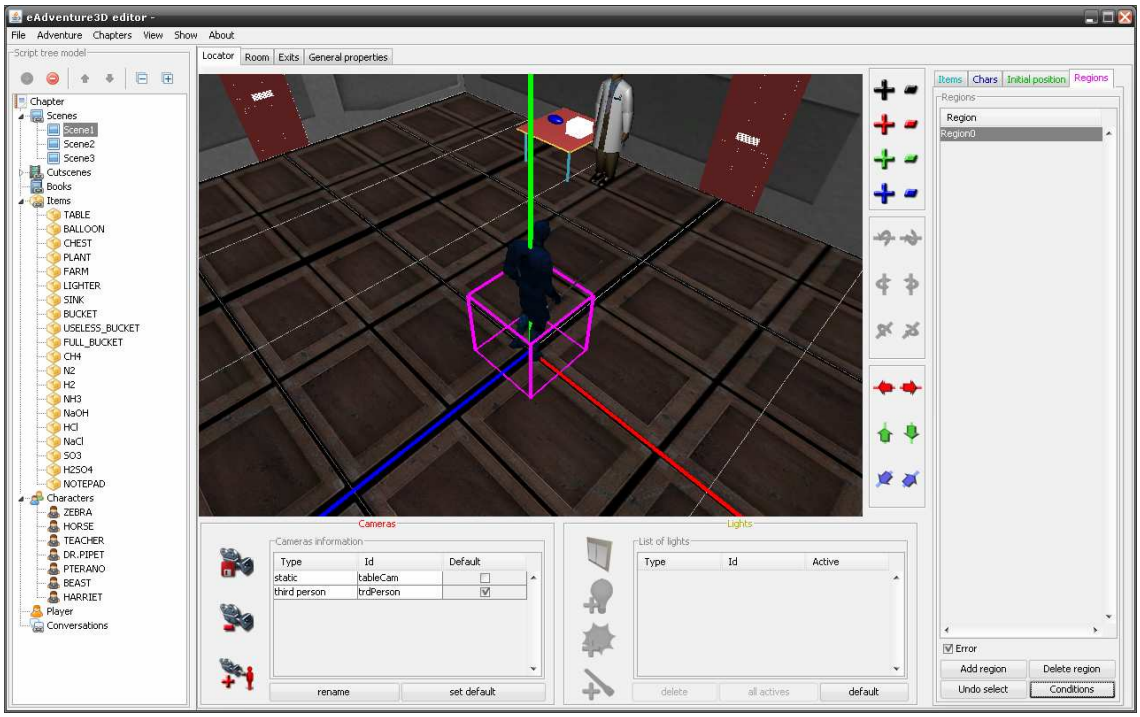
Hemos creado una conversación, pero al hacerlo en el nodo de conversaciones solo podrá ser lanzada como efecto. Para añadir una conversación a un personaje, debemos seleccionar el personaje en el árbol de juego, e ir a la pestaña "Conversations" del mismo. Aquí debemos crear la conversación que queramos de la forma explicada con anterioridad, y al acabar ya estará vinculada al personaje elegido.

El jugador

El jugador representa el personaje manejado por la persona que utiliza el juego. Este es el personaje que interactúa con el resto de personajes u objetos. Los campos que podemos editar aquí son los mismos que para un personaje, a excepción de la pestaña "Conversations" que no tiene sentido que aparezca. Necesitamos modelos con animaciones asociadas.

2.5. Las regiones

Las regiones son zonas de las escenas que podemos definir en <e-Adventure3D>. Empezamos creando una región. Para ello nos vamos a la pestaña "Regions" que nos aparece en la pestaña "Locator" en la escena en cuestión en la que queremos crear la región. Pinchamos en el botón "Add" y seleccionamos la región recién creada que nos aparecerá en una lista:



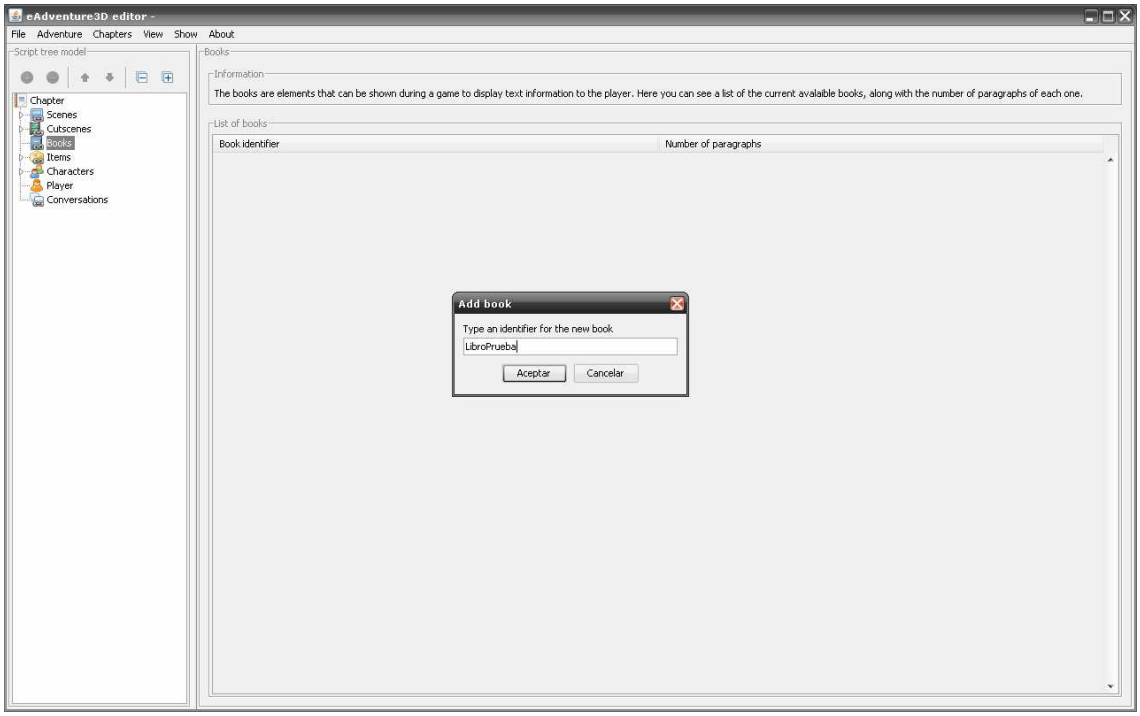
Ahora podemos decidir que tipo de región queremos crear. Si la región es atravesable marcaremos la casilla de abajo. En este tipo de regiones podemos editar las condiciones que permitirán (si se cumplen) o no (si no se cumplen) que cuando pasemos por esa región se lancen los efectos asociados. Podemos añadir cualquier efecto que se ejecutará cuando entremos en la región y se cumplan las condiciones. También podremos añadir postefectos, los cuales se ejecutarán cuando salgamos de la región (veremos todo lo relacionado a efectos y condiciones en la sección 3). Si no marcamos la casilla de abajo, indicamos que estamos creando una región no atravesable. Estas regiones son muy útiles para delimitar el espacio por el que se podrá mover el personaje principal.

La región puede ser editada como cualquier objeto o jugador, pudiendo usar los iconos de desplazamiento, escalado y rotación.

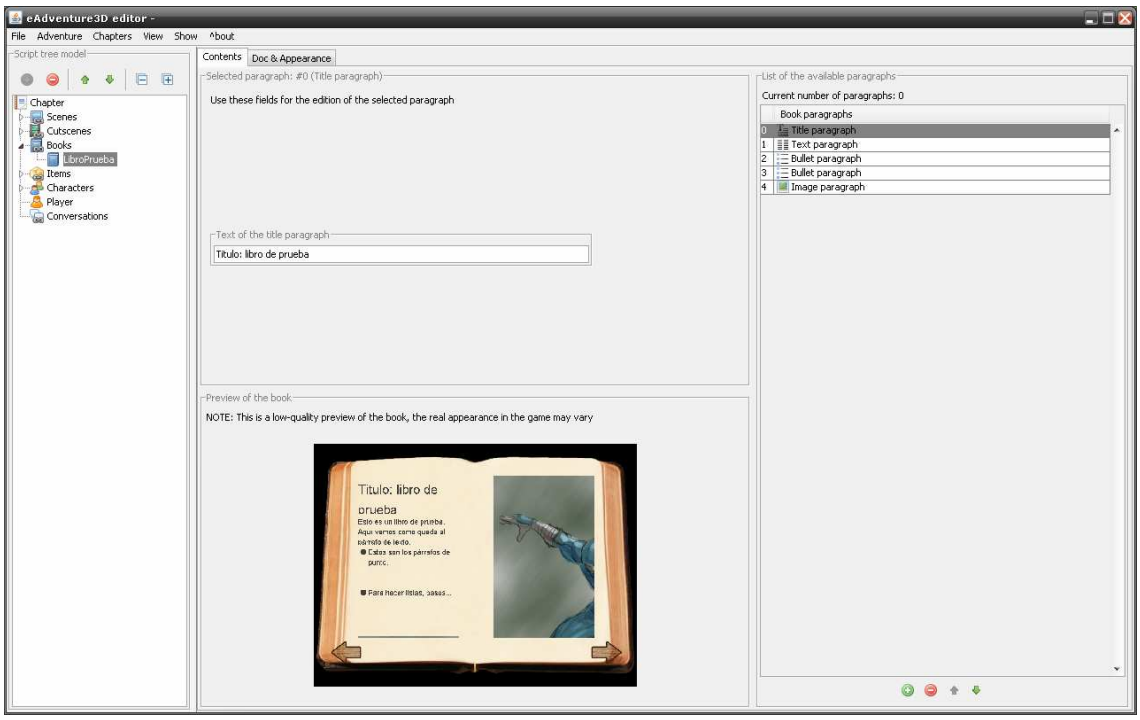
2.6. Los libros

Las conversaciones son una buena fuente de información, como ha sido expuesto con anterioridad. No obstante, <e-Adventure3D> esta concebido para producir videojuegos educativos, por lo que se requerirá en ocasiones proveer de grandes cantidades de información. Para ello podemos incluir libros en los juegos <e-Adventure3D>. Estos libros estarán disponibles donde y como especifiquemos, y pueden contener una sucesión de párrafos de texto, puntos e imágenes.

Vamos a dar una explicación mejor por medio de un ejemplo. Vamos al panel de la izquierda y seleccionamos el nodo “Books”, y añadimos un libro de la misma forma que lo hemos hecho para otros elementos (pulsando el botón derecho sobre el nodo). Al pulsar sobre el nodo “Books” nos aparecen todos los libros ya creados con el número de párrafos que tienen. Lo primero que nos es solicitado al añadir un libro es su identificador:



Una vez elegido el nombre, pasamos a editar las características principales de los libros. En la pestaña “Contents” es dónde vamos a crear los párrafos, elegir su nombre, y ver una previsualización orientativa de cómo va a quedar el libro.



Creando nuevos párrafos

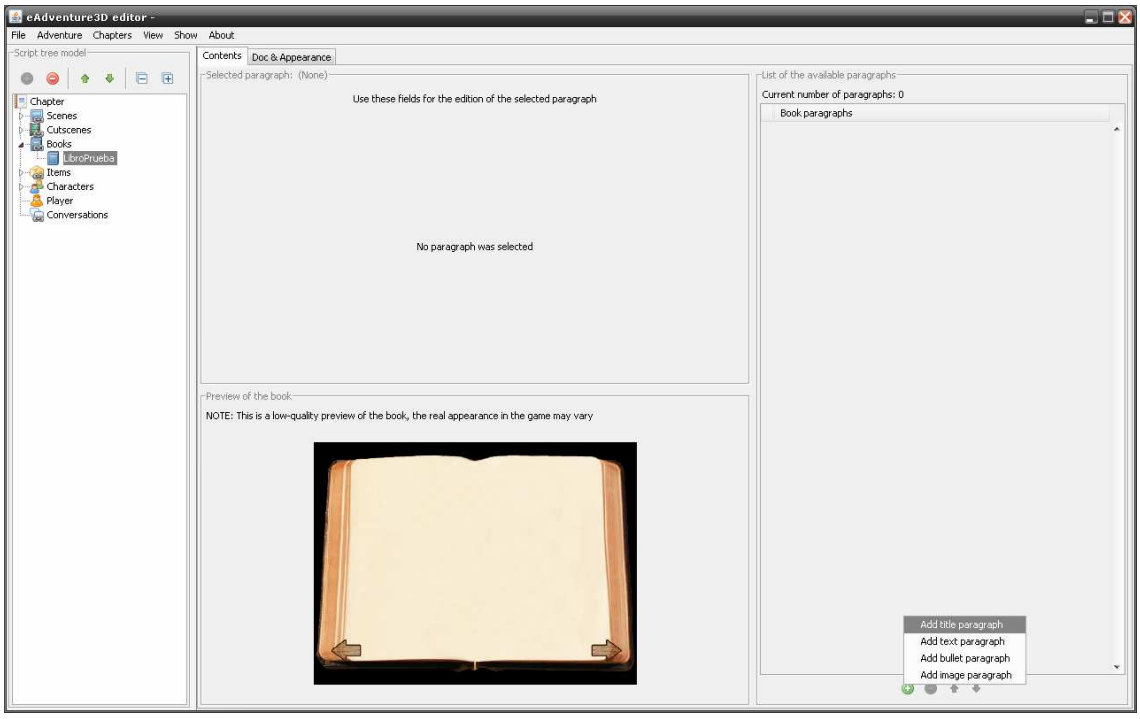
Vamos a explicar como añadir párrafos. Podemos añadir tantos párrafos como

consideremos necesario, y el motor se encargará de repartirlos en varias hojas cuando no quepan en una sola. Para añadir un párrafo no tenemos más que pinchar en el icono  que aparece en la parte inferior derecha de la pestaña "Contents".

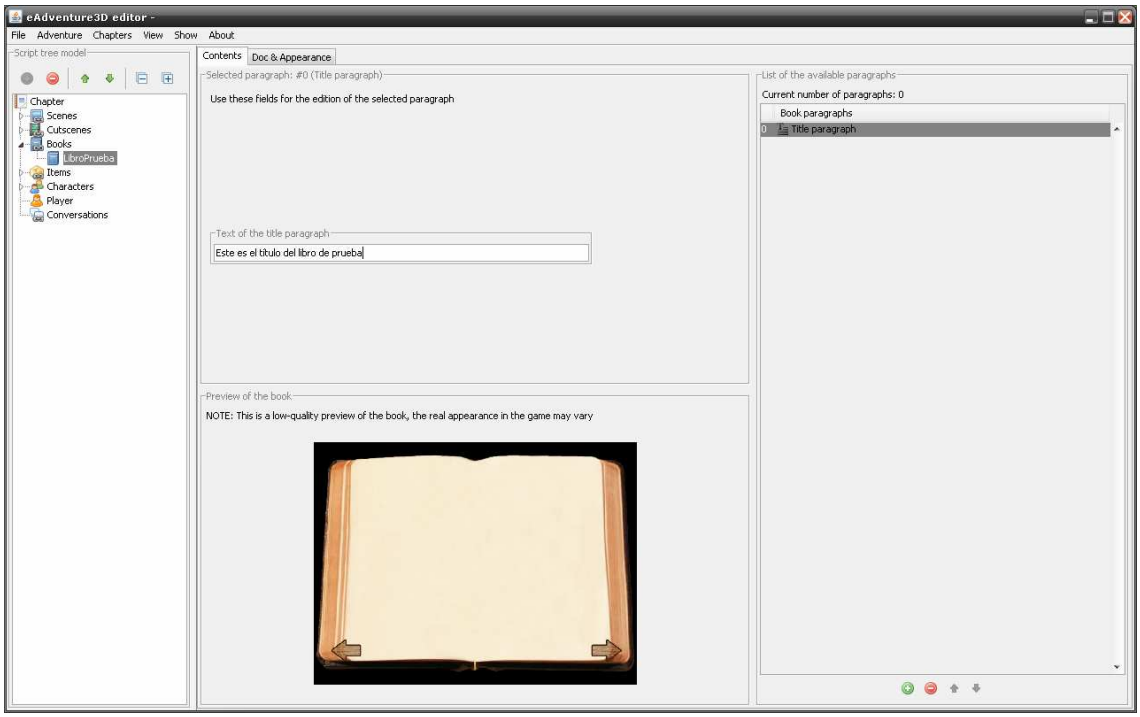
Por ejemplo, vamos a crear un libro con un párrafo de texto y una imagen. También podríamos crear un párrafo de título, el cual es igual que uno de texto pero se renderizará usando un tamaño mayor de letra para "hacerlo mas importante". Hay otro tipo de párrafo soportado por <e-Adventure3D>: párrafo de punto. Este será muy útil cuando queramos organizar un texto en varios puntos.

Comenzamos añadiendo un nuevo párrafo de título. Por ejemplo, pondremos como texto del párrafo de título "Este es el título del libro":

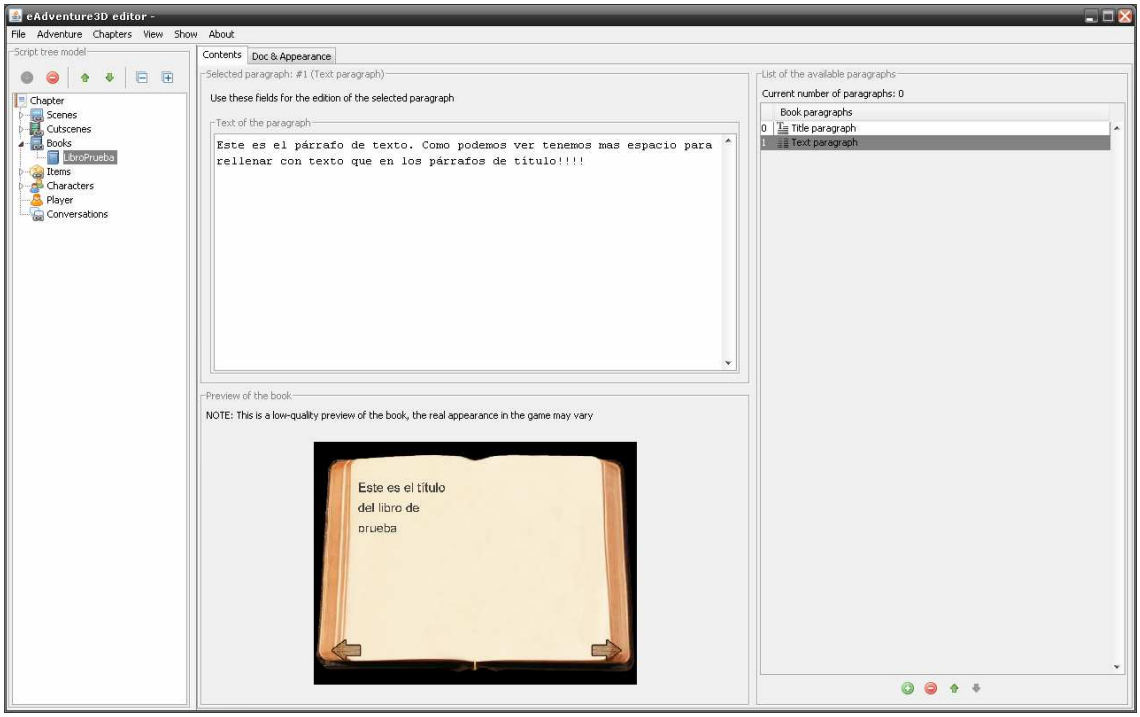
Creamos el párrafo de título:



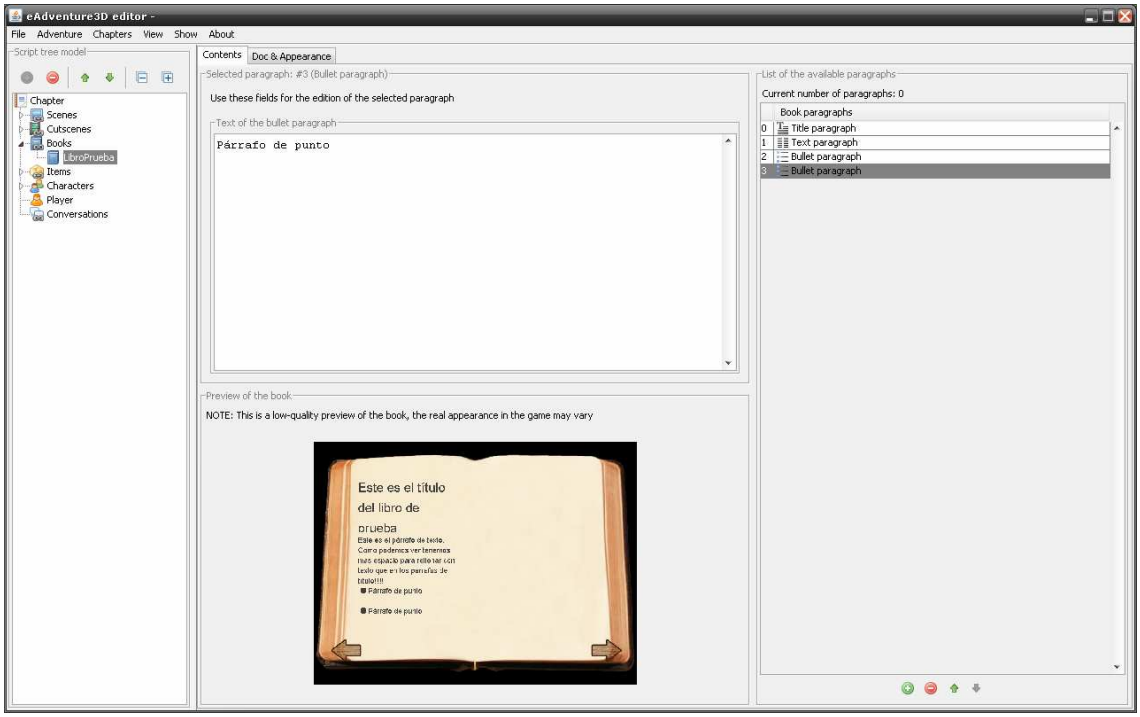
Lo seleccionamos en la lista de párrafos que nos aparece en la derecha, y rellenamos el campo "Text of the title paragraph" con el texto deseado para el título.



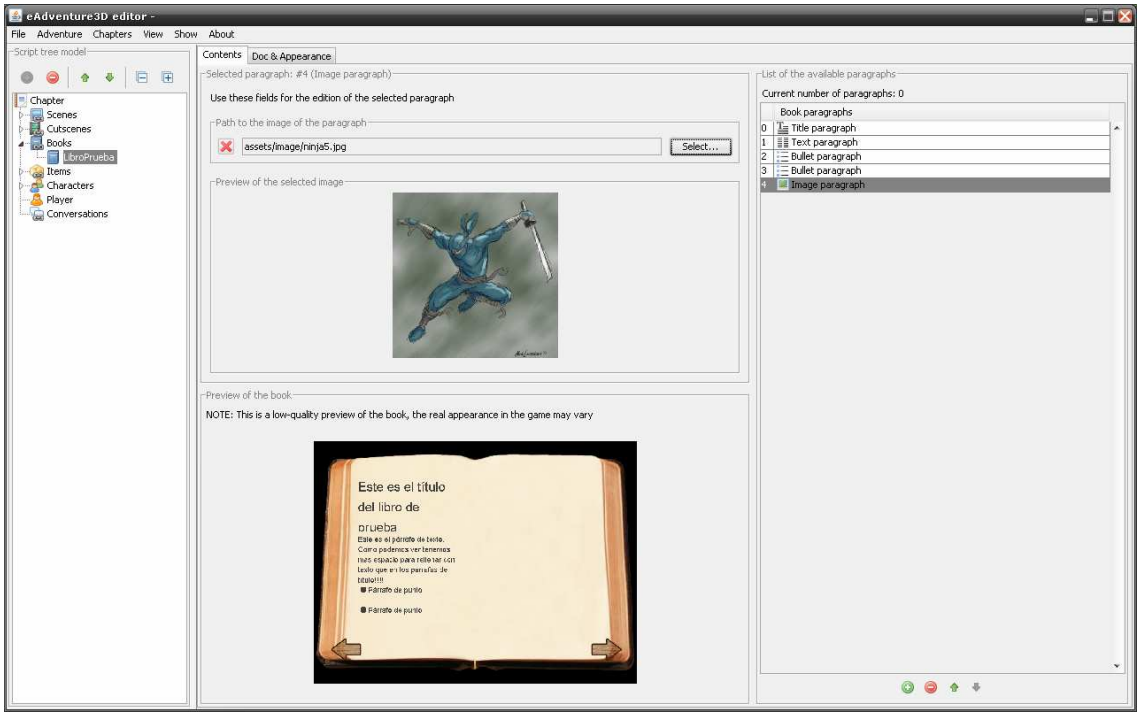
Luego creamos un párrafo de texto, de igual manera que hemos hecho con el de título:



También podemos crear párrafos de tipo “bullet” o punto, los cuales llevan un punto delante para enumerar, poner pasos…:



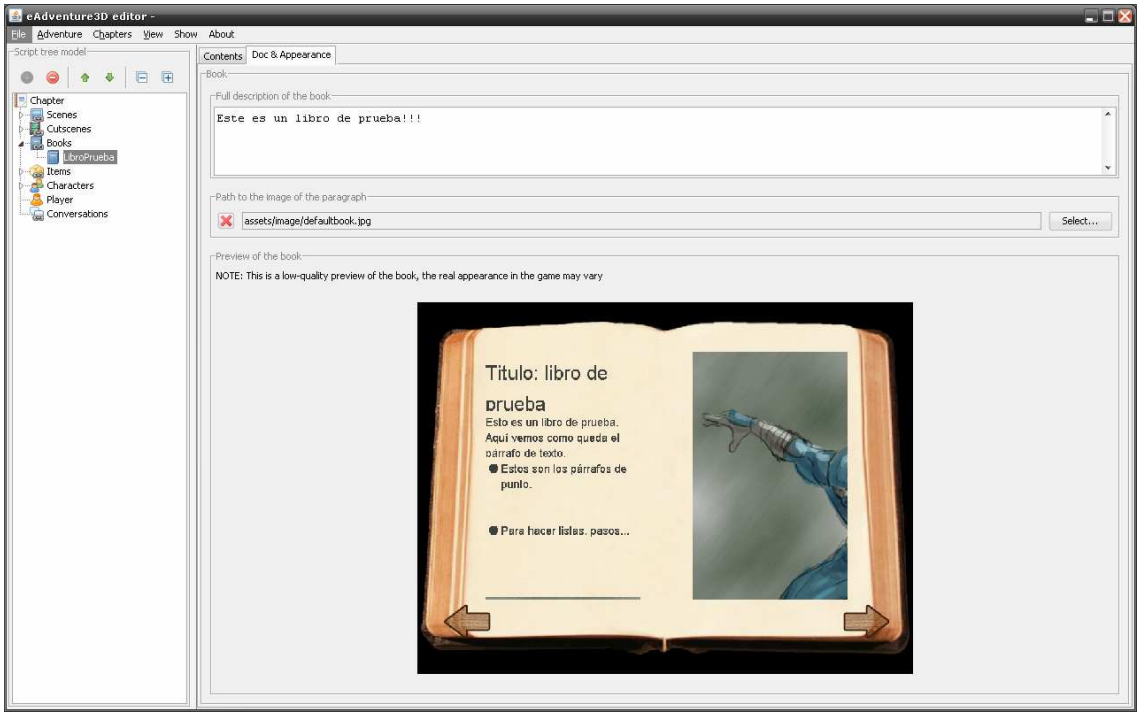
Por último, vamos a añadir un párrafo de imagen. Debemos elegir la ruta de la imagen a añadir en el recuadro “Path to the image of the paragraph”:



Para organizar los párrafos podemos utilizar las flechas que tenemos junto a los

botones de añadir/eliminar párrafos.

Finalmente, seleccionamos la pestaña “Doc & Appearance” para ver la previsualización mas grande, y para rellenar la descripción del libro:



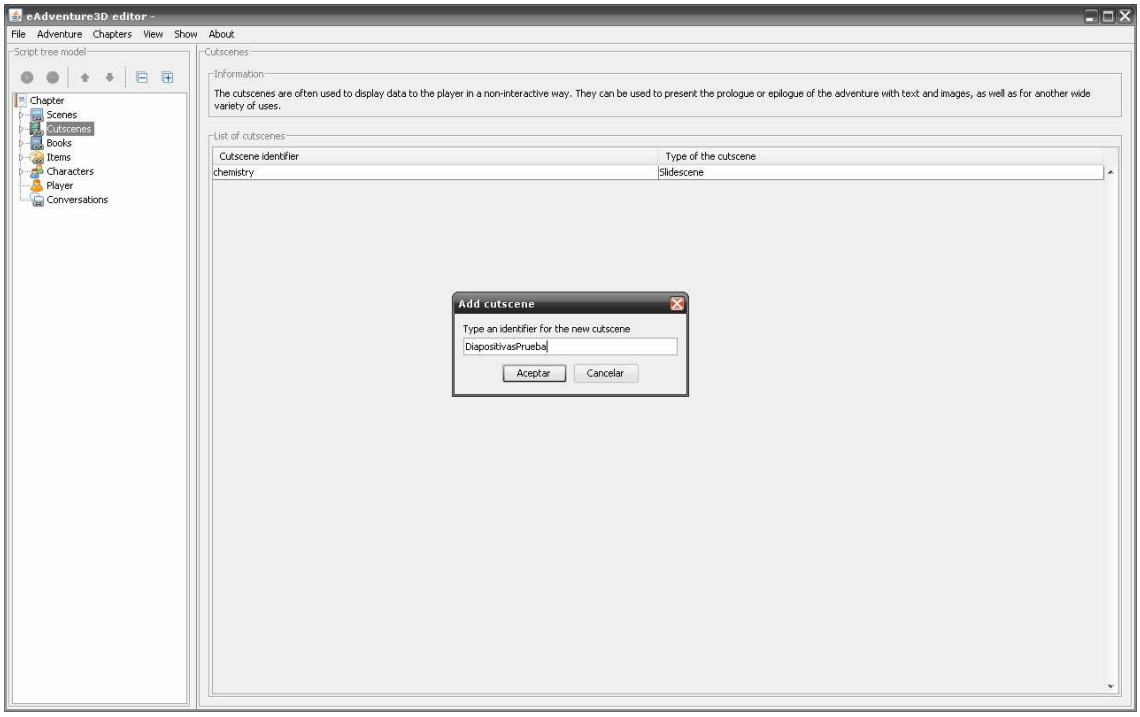
2.7. Escenas de corte (Cut-scenes)

Como hemos visto al principio de este tutorial, los juegos <e-Adventure3D> estan organizados en escenas en donde tiene lugar la acción. Sin embargo, existen otro tipo de escenas, las cuales tienen un concepto completamente distinto del de “escenario donde están los objetos y personajes”. Estas son las escenas de corte (“Cut-scenes”), partes de documentos de información mostrados a pantalla completa cuando el jugador los localiza, en un cambio de escena o al empezar o acabar la aventura, y que pueden conducir a otras escenas o escenas de corte cuando terminan.

Hay dos tipos de escenas de corte: diapositivas (“Slide-scenes”) y escenas de vídeo (“Video-scenes”). Como puedes imaginar, las primeras son un conjunto de imágenes que aparecen unas de tras que otras, y las otras son simplemente un video en formato MPG, MOV o AVI.

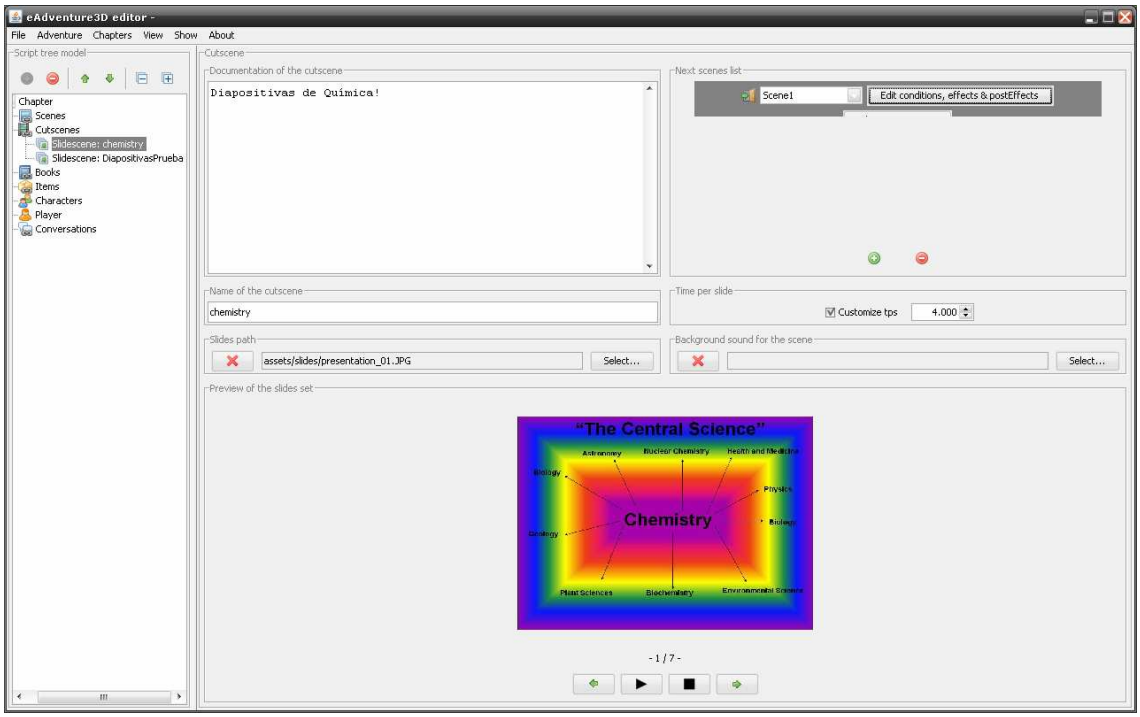
Diapositivas (Slide-scenes)

Comenzamos creando una nueva escena de diapositivas. Es tan sencillo como crear nuevos elementos como lo llevamos haciendo hasta ahora. Por ello, pinchando con el botón derecho en el nodo “Cutscenes” y seleccionando “Add slidescene” crearemos una nueva escena de diapositivas. De igual manera, si pinchamos con el botón izquierdo sobre el nodo “Cutscenes” nos aparecen todas las escenas de diapositivas existentes, con el tipo de escena que es. Como viene siendo habitual, se nos requerirá el identificador para este tipo de escena:



Una vez seleccionado el nombre, nos aparece la ventana en la que podremos configurar las opciones de las diapositivas. Podemos rellenar la documentación asociada a las diapositivas en el campo de texto (“Documentation of the scene”) y el nombre (“Name of the cutscene”). En “Slide path” elegimos la ruta del conjunto de imágenes que vamos a añadir. Se escoge la primera imagen del conjunto, todas deben tener el mismo nombre y estar numeradas; es decir, si queremos meter tres transparencias las llamaremos: Nombre_01, Nombre_02, Nombre_03; las meteremos en la misma carpeta y seleccionaremos la primera de ellas, así el editor

sabrá que existen las otras 2. Una vez añadidas, podemos ver la previsiualización, pasando las diapositivas con las flechas, o viendo como va a quedar pinchando en el icono de “play”. Además podemos seleccionar el tiempo que va a durar cada transparencia en fps (en el recuadro “Time per slide”). En el recuadro “Next scene list” podemos editar las condiciones y efectos asociados a las diapositivas; seleccionar la siguiente escena (o escenas), una vez acaben las diapositivas; seleccionar que la escena a de diapositivas va a ser final del capítulo.

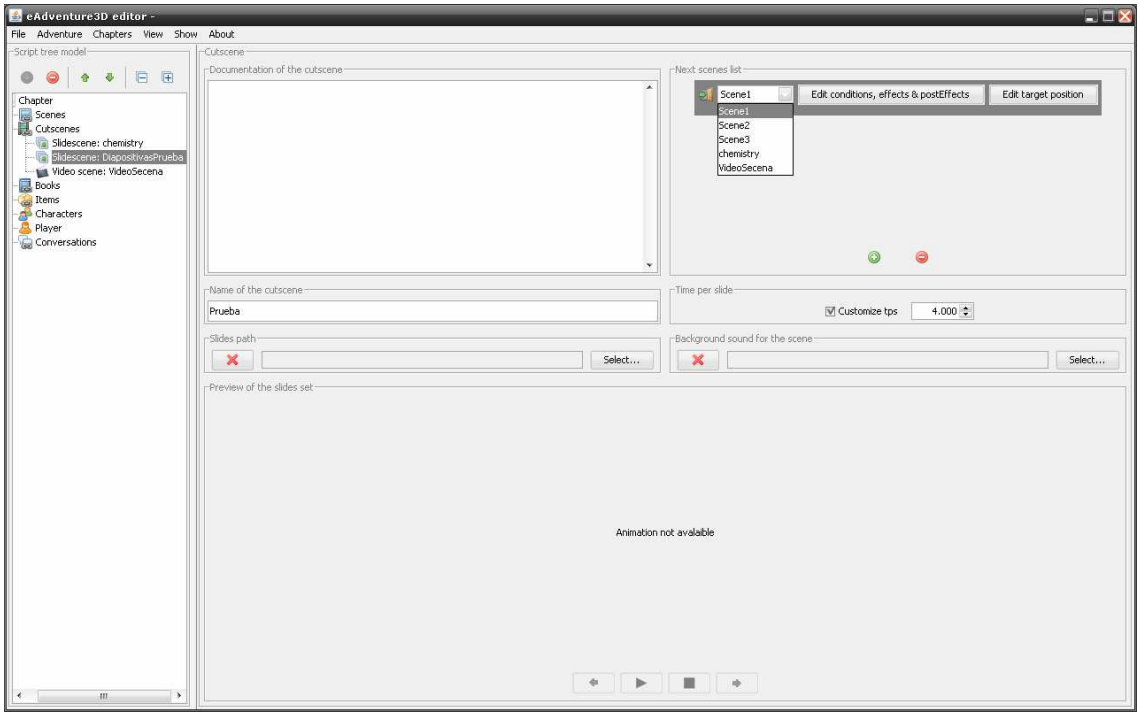


Video-scenes

Creamos y editamos una nueva escena de video de igual manera que las escenas de diapositivas. Simplemente seleccionamos el nodo “Cutscenes” y seleccionamos “Add videoscene”. El panel es el mismo que aparece para las escenas de diapositivas (excepto porque en las escenas de video no permitimos meter sonido de fondo porque se presupone que el propio video lo incluirá). Solo necesitamos elegir la ruta del video que deseamos añadir pinchando el botón “Select...”.

Uniendo escenas de corte a escenas

Como con las escenas, las escenas de corte pueden ser unidas a otras escenas o escenas de corte. Este es un comportamiento esencial ya que una escena de corte sin uniones no sabrá por donde continuar una vez acabada, por lo que bloqueará el juego. Este proceso es similar al de unir escenas, lo único es que no tenemos que hablar de salidas. Simplemente tenemos que editar el campo “Next scene list” del la escena de corte en cuestión. Elegiremos “Add next scene” para ir a una escena o escena de corte cuando acabe la escena de corte seleccionada. Elegiremos “Add end scene” si queremos que después de la escena de corte se acabe el capítulo. Por ejemplo, unimos nuestra escena de diapositivas a la primera escena del juego, de manera que después de las diapositivas empezará el juego realmente. Simplemente seleccionamos “Add next scene” y la escena a la que queremos movernos.



Podemos añadir tantas salidas como veamos necesario. El motor decidirá donde ir analizando las condiciones para todas las escenas añadidas. La primera escena que satisfaga todas las condiciones será la usada.

3. Perfeccionando nuestro primer juego <e-Adventure3D>

Llegado a este punto tenemos el conocimiento necesario para crear juego <e-Adventure3D>. No obstante, <e-Adventure3D> soporta más características, aumentando las capacidades de los juegos <e-Adventure3D>. En esta sección vamos a aprender a usar estas características.

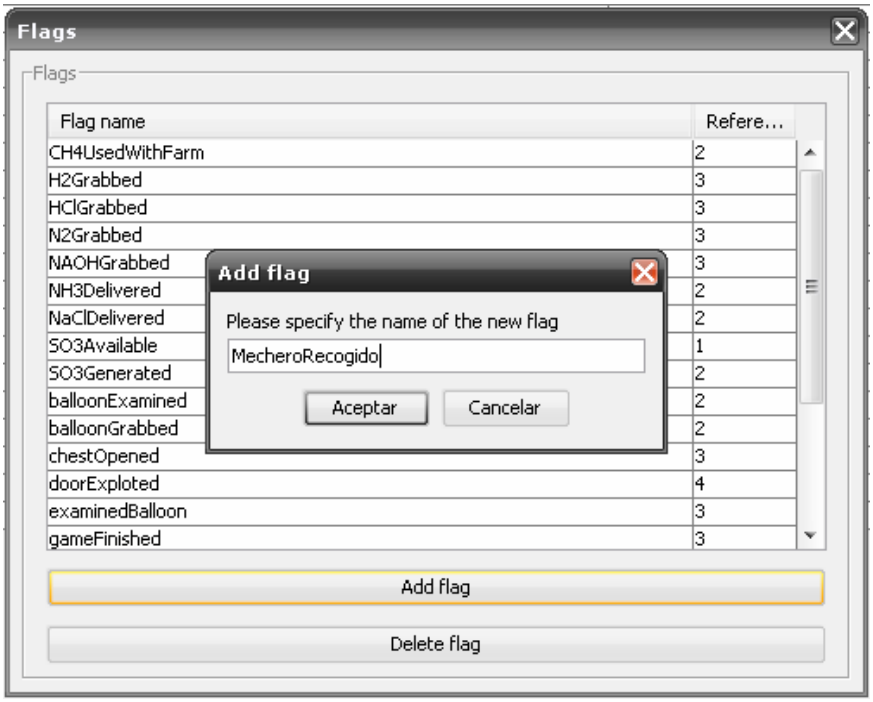
3.1 Condiciones y efectos

Ya aprendimos a crear y editar elementos de un juego <e-Adventure3D>. Sin embargo, un par de objetos, personajes y escenas no hacen un juego. Para hacerlo necesitamos dirigir la historia del juego, llevando a cabo un comportamiento narrativo. En <e-Adventure3D> la historia es dirigida por medio de flags. Estos flags son cadenas de caracteres definidos por el diseñador del juego que pueden ser evaluados en cualquier momento como "cierto" o "falso". De esta forma podemos definir flags, y establecer condiciones sobre esos flags. Vamos a introducirnos en este aspecto a través de un ejemplo.

Condición simple

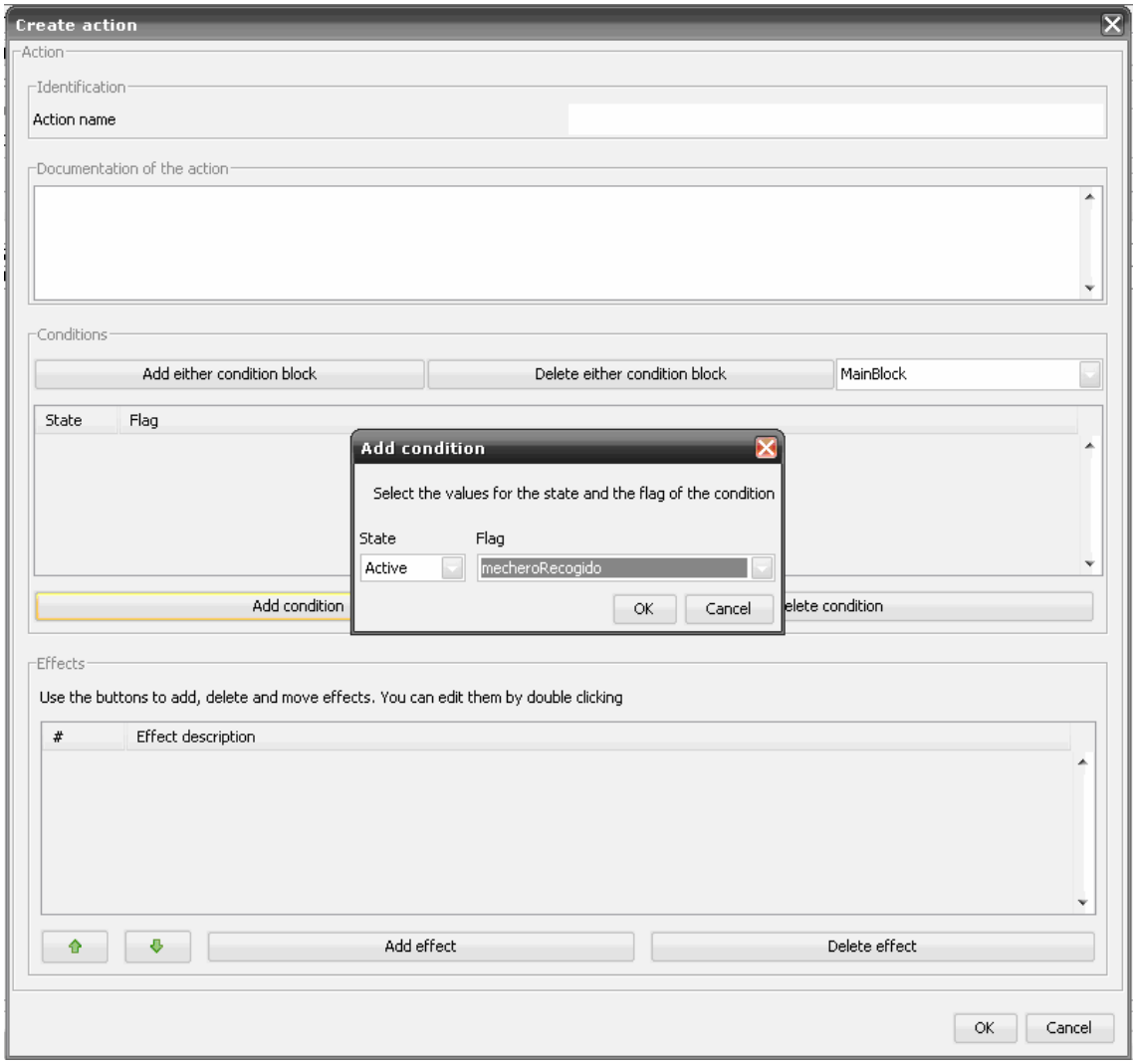
Suponemos que hemos definido un juego que tiene dos objetos: un mechero y una planta. La cuestión es que queremos que el jugador pueda coger la planta solo después de haber cogido el mechero. En nuestra aventura dicha condición es "trasladada" al lenguaje de flags: debemos definir un flag "mecheroRecogido", el cual es falso hasta que el mechero sea recogido. Desde ese momento el flag podrá cambiar su valor a cierto, y por tanto la planta será accesible por el jugador. Vamos a analizar el proceso entero desde el principio:

Primero, necesitamos añadir un nuevo flag. Para hacer esto, vamos al menú "Chapters" de la barra de menú. Aquí seleccionamos la opción "Edit chapter flags". También podemos efectuar esta operación presionando Crtl+F. Aparecerá una ventana como la mostrada a continuación, indicándonos todos los flags existentes en este capítulo, junto con sus referencias en la aventura. Pulsamos el botón "Add flag" y escribimos "mecheroRecogido". Entonces pulsamos "ok" y un nuevo flag con 0 referencias será creado:



Podemos usar esta ventana para añadir o borrar flags. No obstante, si queremos borrar un flag, no debemos preocuparnos por las referencias que este tenga, ya que el editor se encargará de eliminarlo de toda la aventura. Además, fijémonos que la definición de flags tiene algunas restricciones que debemos respetar: no están permitidos los espacios en blanco y el primer carácter debe ser una letra.

Una vez que el flag ha sido creado, podemos hacer referencias al mismo. Podemos ir al objeto planta y añadirle una nueva acción “coger”. Recordatorio: Seleccionamos el nodo del objeto correspondiente en el árbol > Seleccionamos la pestaña “Actions” > Pulsar el botón “Add” de la parte inferior > Seleccionar el tipo de accion (Grab). Después de esto, seleccionamos un bloque de condiciones y pulsamos en “Add condition”. Podemos elegir entre una lista el flags deseado y elegir el estado en el que deberá estar dicho flag (activo o inactivo). Si seleccionamos activo (“active”), la condición se hará verdadera cuando el flag este a “cierto”. Si seleccionamos inactivo (“inactive”) la condición se hará verdadera cuando el flag sea “falso” (por defecto todos los flags están inactivos). En nuestro caso, seleccionamos “activo” y el flag “mecheroRecogido”.



Either conditions

Como habrás visto, en el recuadro de condiciones nos aparecen cuatro botones. Dos son obvios, añadir y borrar condición. Los otros dos, etiquetados como “Add either condition block” y “Delete either condition block”, añadirán una nueva cuenta a continuación de la actual (bloque actual “Main block”). Cada cuenta es un bloque de condiciones que deben ser ciertas para considerar que el bloque es cierto. La condición será cierta cuando al menos uno de los bloques sea cierto. Esto es útil para crear condiciones dependientes de varios flags. Si alguno de los flags debe ser activo o inactivo al mismo tiempo, debe estar localizado en el mismo bloque (condiciones “and” o “y lógica”). Si solo uno de ellos debe estar activo o inactivo, podremos añadirlos en bloques distintos (condiciones “or” u “o lógica”).

Que podemos hacer con condiciones

Ahora que sabemos como definir condiciones, podemos usarlas en casi todos los contextos, cambiando el comportamiento del juego.

Para acciones (objetos).

Para conversaciones (personajes).

Para escenas:

Para cambiar a donde iremos cuando interactuamos con una salida (podemos definir varias siguientes escenas para una salida, y la primera que cumpla las condiciones será la elegida).

Para referencias de objetos y personajes. Por ello, a los personajes y objetos presentes en una escena se les permitirá cambiar durante el juego.

Activando y desactivando flags: efectos

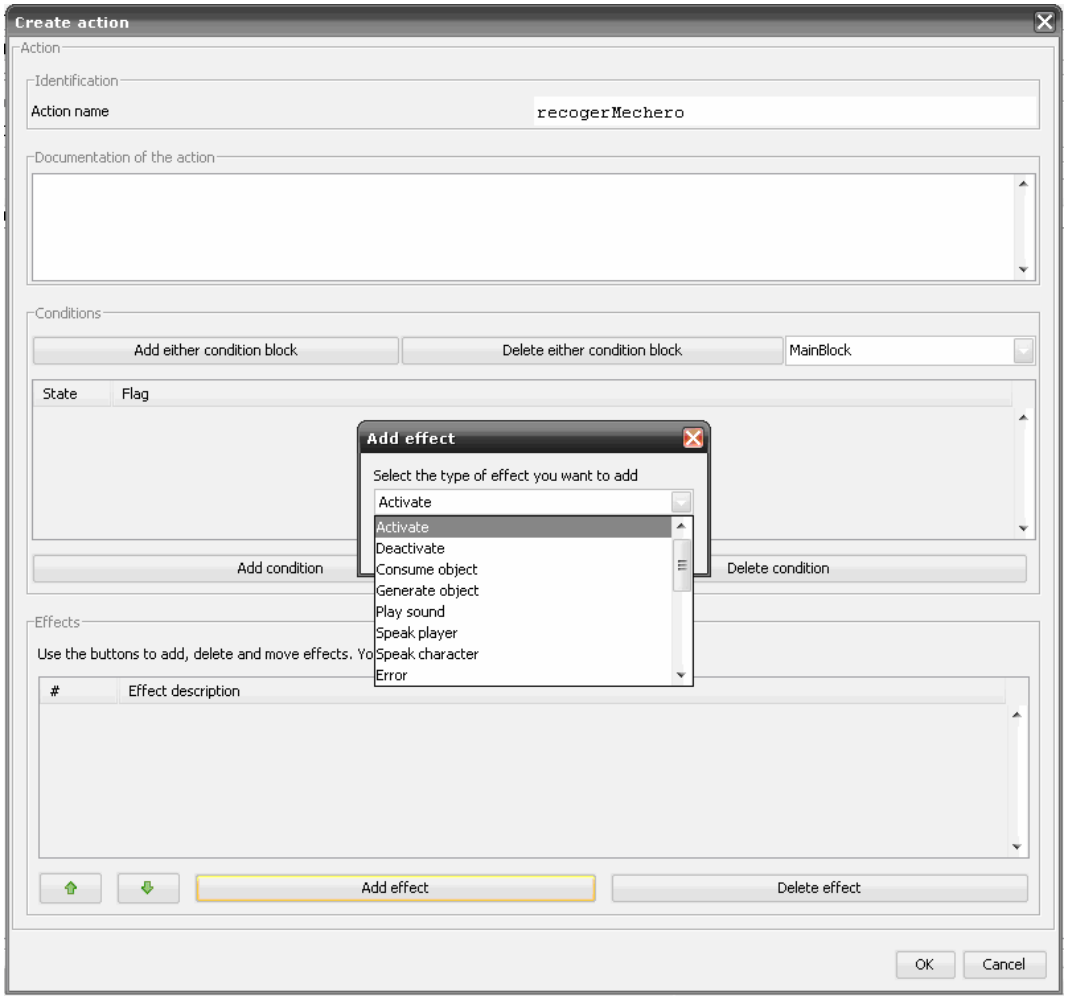
Como has podido observar, definir condiciones en los juegos en una herramienta muy útil. Pero, ¿cómo podemos hacer cambios en los flags? El hecho de establecer condiciones no produce ningún efecto ya que dependen de flags que nunca van a cambiar. En <e-Adventure3D> podemos activar o desactivar flags usando efectos. Podemos definir estos efectos en las siguientes situaciones:

Para acciones (objetos). Cuando una acción es llevada a cabo, sus efectos serán ejecutados.

Para conversaciones (personajes). Podemos definir efectos en cualquier línea de conversación de cualquier nodo de una conversación. De esta manera, cuando la línea es alcanzada, se lanzan los efectos. También podemos asociar efectos a nodos terminales, ejecutándose cuando se acabe las líneas de conversación del mismo.

Para escenas. Podemos producir efectos cuando la escena actual cambia. En este caso podemos definir efectos, los cuales serán ejecutados antes de que la escena cambie, y post-efectos, los cuales se ejecutarán una vez la escena haya cambiado.

Siguiendo con nuestro ejemplo, si queremos que el flag mecheroRecogido se active después de que el mechero haya sido recogido, debemos añadir la acción “coger” al objeto y editar sus efectos:



Para conseguir esto, seleccionamos la pestaña “Actions” del objeto en cuestión y pinchamos en el botón “Add effect” de la parte inferior de la ventana. Nos pedirá el tipo de acción a la que vamos a añadir el efecto, y el tipo de efecto a lanzar. Para nuestro ejemplo elegiremos la acción “grab” y el efecto “activate”. Luego nos pide elegir el flag que vamos a activar con el efecto, seleccionando el efecto “mecheroRecogido”.

Los efectos a fondo

Los efectos son usados cuando queremos activar o desactivar flags. Sin embargo, <e-Advenure3D> soporta más tipos de efectos. Aquí podemos encontrar todos los efectos soportados con una descripción para cada uno de ellos y sugerencias sobre como usarlos:

Efecto	Descripción	Usos
Consumir objeto (Consume item)	Elimina un objeto de una escena	Un uso típico para este efecto es hacer desaparecer los objetos cuando son cogidos.
Generar objeto (Generate item)	Pone un objeto en el inventario	Típicamente usado cuando se coge un objeto
Cancelar acción (Cancel action)	Cancela la acción por defecto	
Hablar jugador (Speak player)	Hace que un jugador diga una única línea de conversación.	Podemos usar este tipo de efectos cuando Se prohíbe una acción.
Hablar personaje (Speak character)	Hace que un personaje diga una única línea de conversación.	Podemos usar este tipo de efectos cuando se prohíbe una acción.
Lanzar libro (Trigger book)	Muestra uno de los libros definidos en el juego.	Este efecto se usa para proveer información en forma de libro, hasta que el usuario decida salir del mismo.
Reproducir sonido (Play sound)	Reproduce el sonido seleccionado.	Puede ser usado para producir efectos de sonido para acciones, ya que dichos sonidos solo se reproducen una vez.
Mover jugador (Move player)	Hace que el jugador valla a la posición seleccionada (x,y).	
Mover personaje (Move character)	Hace que el personaje seleccionado valla a la posición seleccionada (x,y).	Un típico comportamiento en aventuras gráficas: personaje y jugadores se mueven mientras hablan.
Lanzar conversación (Trigger conversation)	Lanza la conversación seleccionada.	Podemos usar este efecto para proveer una guía interactiva en algunos puntos del juego.
Lanzar escena de corte (Trigger cutscene)	Lanza la escena de corte seleccionada.	Un video o una secuencia de diapositivas reproducida para proveer información o explicaciones en algunos puntos del juego.
Lanzar escena (Trigger scene)	Cambia la escena actual	Muy útil ya que permite cambiar de escena sin interactuar con una salida.
Cambiar animaciones jugador (Set player animations)	Cambia la animación (conjunto de frames) del modelo para el jugador	Útil si queremos que el jugador realice un movimiento fuera de los usados por defecto.
Cambiar animaciones personaje (Set npc animation)	Cambia la animación (conjunto de frames) del modelo para un personaje	
Cambiar luz (Change light)	Enciente la luz escogida si estaba apagada y la apaga si estaba encendida	Nos puede servir para ocultar a la vista partes de escena que no queramos enseñar

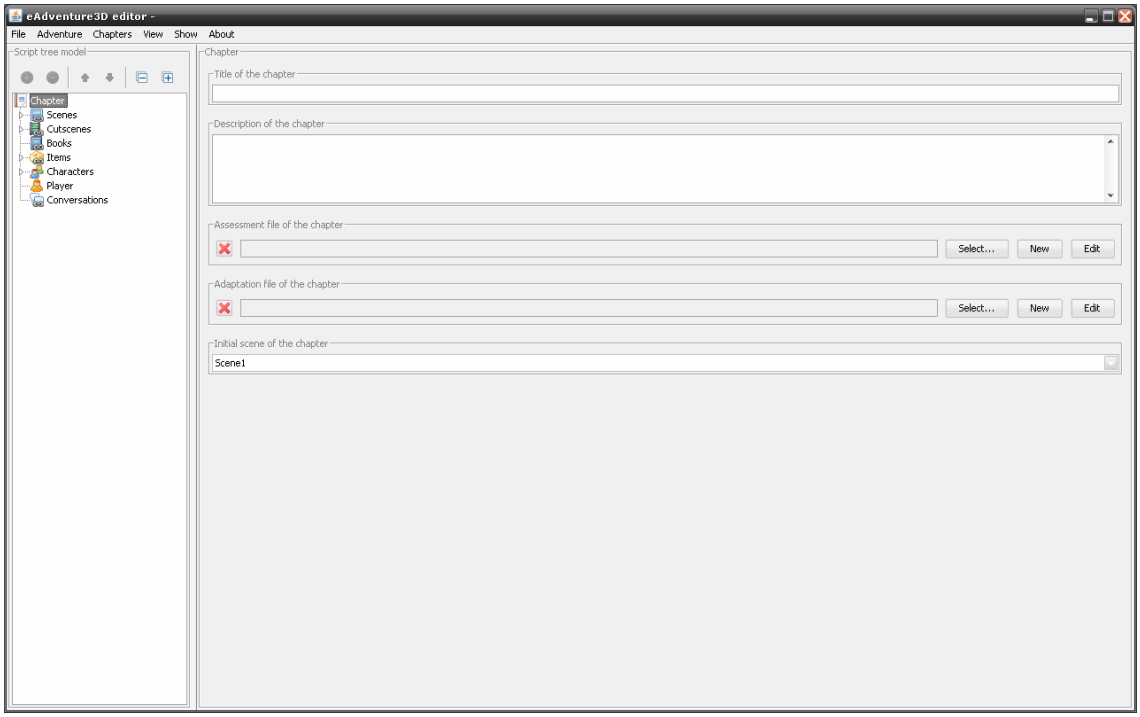
		en un determinado momento.
(Change camera)	Cambia la cámara.	Muy utilizada cuando queremos remarcar de forma visual algún aspecto de la escena (hacer un plano de detalle de un objeto, personaje...).
Finalizar capítulo **(End chapter)**	Finaliza el capítulo o termina la aventura si no hay más capítulos.	Necesario para cambiar de capítulo.

3.2. Características de adaptación y evaluación

En las secciones anteriores hemos aprendido todo lo necesario para crear un juego <e-Adventure3D>. Sin embargo, <e-Adventure3D> esta especialmente diseñado para la producción de videojuegos educativos. De acuerdo con las actuales tendencias en la educación on-line, estos objetos educativos deben ser valorados y adaptados de acuerdo con las características del jugador (aprendiz).

De ahí que <e-Adventure3D> soporte características tanto de adaptación como de evaluación. En esta sección vamos a aprender a editar dichas características.

Las características de adaptación y evaluación están separadas del storyboard del juego. Por eso el mismo juego puede ser usado en diferentes contextos educativos, cambiando los ficheros de adaptación y evaluación. Estos ficheros son diferentes para cada capítulo. Para editarlos, seleccionar el nodo “Chapter” en el árbol. Como vemos, hay tres botones para la edición de los ficheros de adaptación y evaluación:



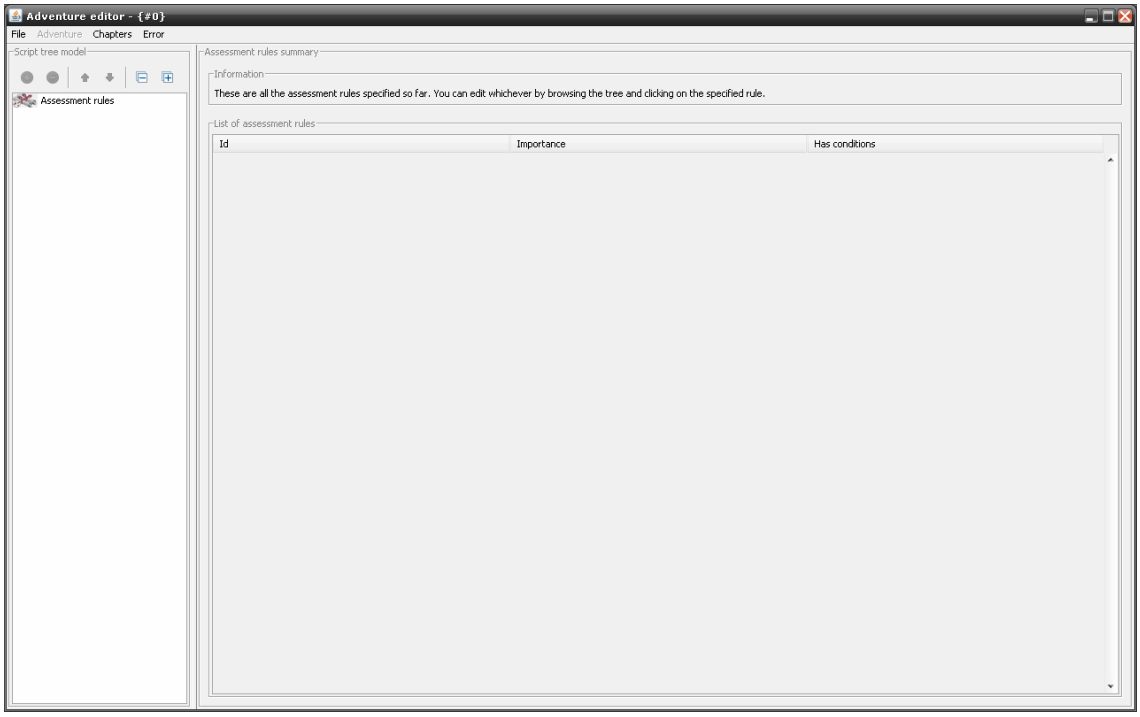
Select: Permite cambiar los ficheros de adaptación/evaluación para el capítulo en cuestión, mirando entre todos los ficheros de adaptación/evaluación adjuntos en el juego.

New: Crea un nuevo fichero de adaptación/evaluación. El editor cambiará presentando las opciones necesarias para editar dichos ficheros.

Edit: Edita el fichero actual de adaptación/evaluación. El editor cambiará presentando las opciones necesarias para editar dichos ficheros.

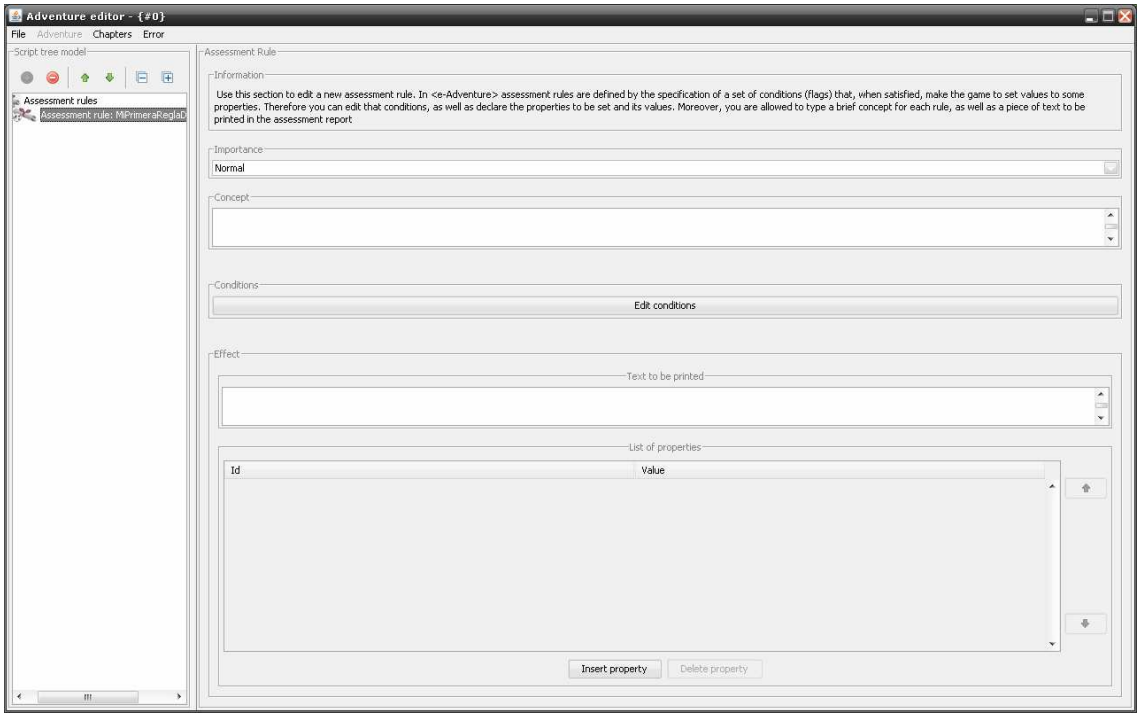
Evaluación

Vamos a crear un nuevo fichero de evaluación para nuestro juego. Para ello, clickeamos en “New”. Después confirmamos que queremos cambiar al modo de evaluación y escribimos el nombre que va a tener el nuevo fichero. Nos aparece una ventana como la siguiente:



Se consigue una evaluación automática especificando reglas de evaluación. El efecto de estas reglas es escribir un informe <html> que es accesible para el instructor después de la ejecución del juego. Usamos el nodo “Assessment rules” para añadir y borrar dichas reglas.

Por ejemplo, vamos a añadir una nueva regla llamada “MiPrimeraReglaDeEvaluacion”. Veremos los siguientes campos editables para la regla de evaluación:



Importance: Es normal por defecto. Admite los siguientes valores: Muy baja (*Very low),* baja *(low),* normal *(normal),* alta *(high)* y muy alta *(very high*).

Concept: La descripción de la regla

Conditions: La regla de evaluación debe ser ejecutada bajo unas condiciones. Esto es esencial porque sin estas condiciones todas las reglas serían siempre ejecutadas. Podemos activar flags para diferentes situaciones de juego, y por consiguiente evaluar al estudiante de distintas formas. Por ejemplo, activando un flag cuando intenta hacer algo demasiadas veces, el cual lanzará una regla de evaluación que lo suspenderá. Por otro lado, cuando el número de intentos sea pequeño, otra regla que apruebe al alumno será lanzada.

Text to be printed: Un fragmento de texto para ser añadido al informe.

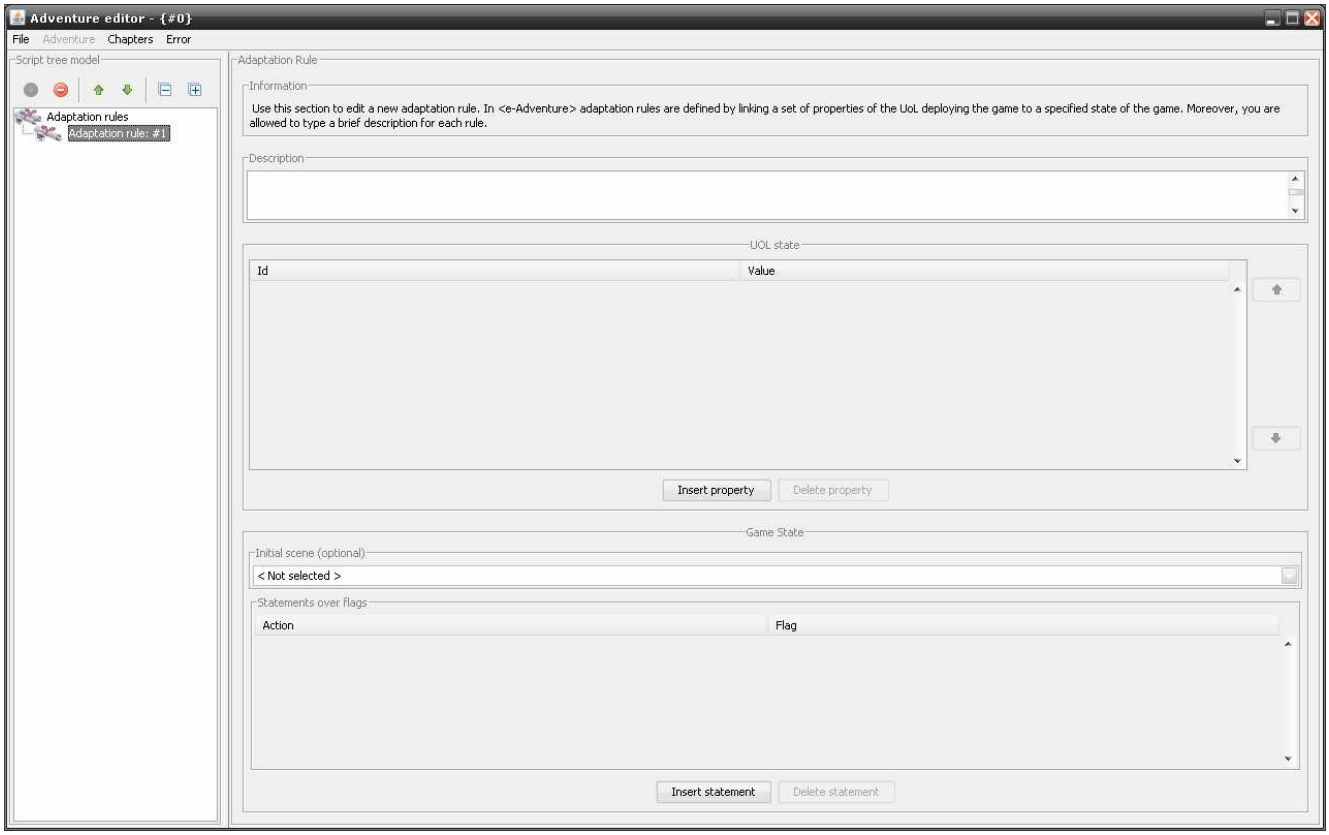
Properties: Cada propiedad es representada como un valor entero. Estos pueden ser usados para fijar notas finales o reducciones parciales de la nota final.

Una vez hemos editado las reglas de evaluación, clickeamos en “Edition Mode > Return to Adventure Mode” para volver a el modo de edición de aventura.

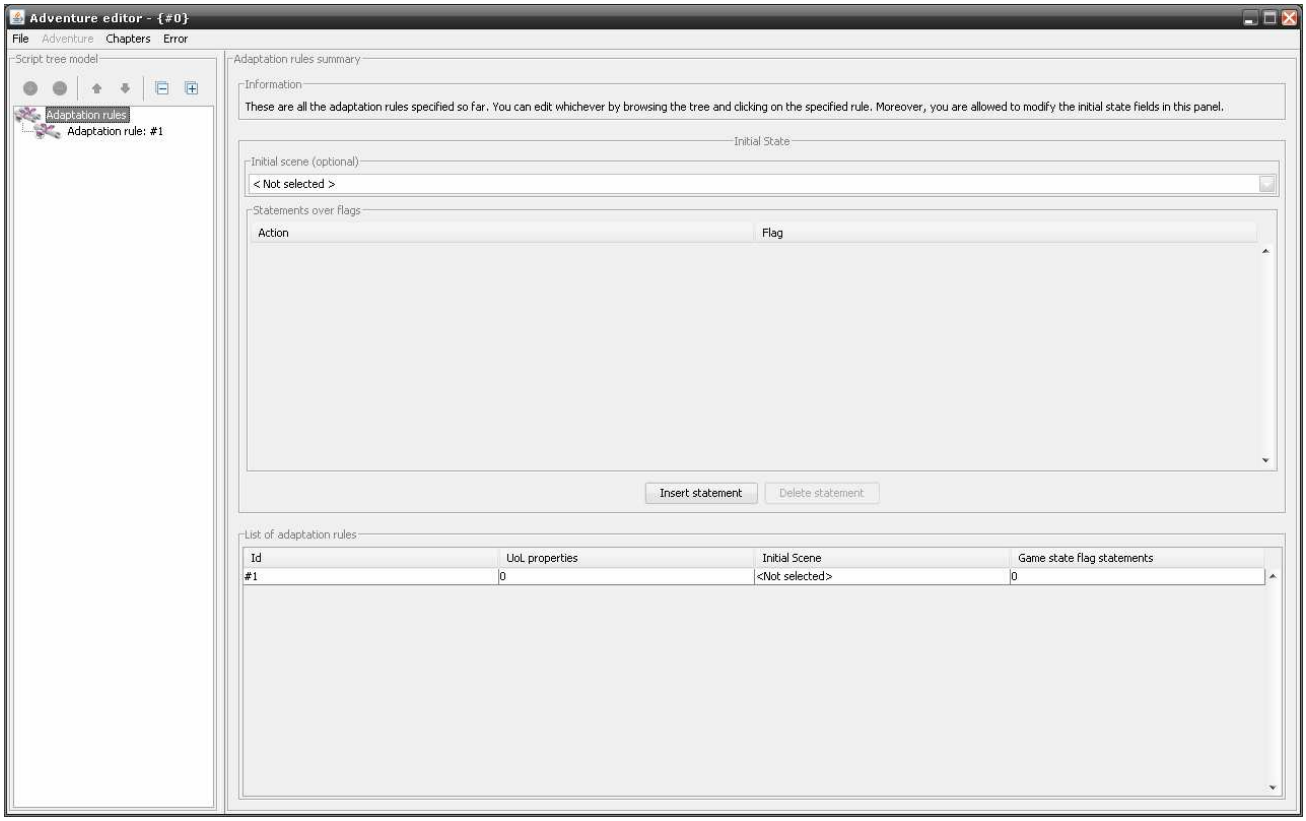
Adaptación

Los ficheros de adaptación son gestionados de igual manera que los de

evaluación. El fichero de adaptación se usa para adaptar el juego de acuerdo con un conjunto de propiedades que viene dado por la Unidad de Aprendizaje la cual utiliza el juego (UoL state). Cuando estas propiedades son satisfechas el estado del juego se adapta activando o desactivando algunos flags y seleccionando una escena inicial. El panel de edición de las reglas de adaptación es el siguiente:



Además, el estado inicial del juego puede ser editado en el nodo principal del árbol que nos aparece en el modo de edición de reglas de adaptación:



3.3 Otras opciones

En esta sección describimos como usar los menús de la barras de menú:

Menu “File”

Este menú provee las opciones convencionales de salvar, cargar y crear un nuevo fichero <e-Adventure3D>

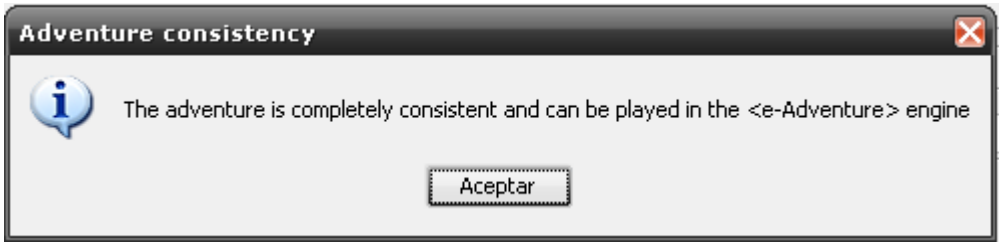
Menu “Adventure”

Este menú provee de opciones de edición de los datos generales de la aventura. Estas opciones son las descritas a continuación.

Confirmar la consistencia de la aventura

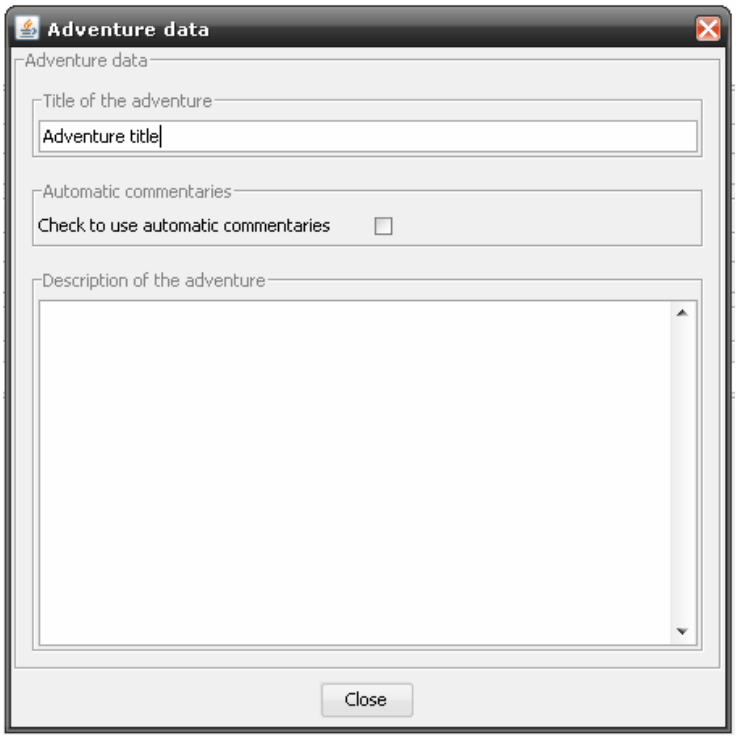
Esta opción del menú genera un informe con todas las incidencias que impedirían que el juego fuese ejecutado. Por ejemplo, un aspecto que invalidará el fichero es que no haya escenas en una aventura. Si todo va bien mostrará este

mensaje:



Editar los datos de la aventura (edit adventure data)

Nos permite editar los datos generales del juego. Esto incluye el título y una descripción de la aventura. Además de la opción de comentarios automáticos. Si está activa, durante el juego se mostrarán comentarios en inglés sobre cosas que no se pueden hacer cuando el usuario trate de hacerlas.



Ficheros de adaptación y evaluación (Manage assesment/adaptation files)

Si seleccionamos en cualquiera de las opciones podemos ver un diálogo donde podemos añadir y borrar los ficheros de evaluación o adaptación. Si los borramos, no serán accesibles nunca más.

Gestión de recursos (Manage assests)

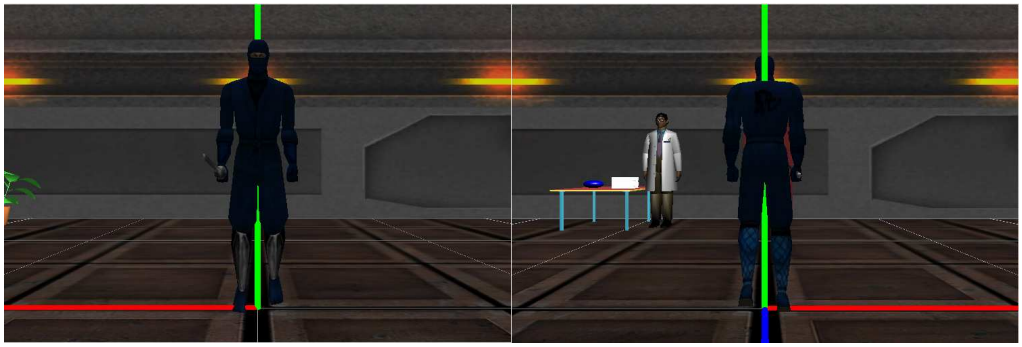
En el menú "Adventure" podemos gestionar los siguientes recursos: imágenes, iconos, texturas, ficheros de audio, de vídeo, diapositivas, "skybox" y modelos. Desde cada una de las opciones podemos añadir o eliminar cada recurso del archivo que compone la aventura que ya no queramos usar.

Menú de capítulos

Usar este menú para cambiar el capítulo actual que estamos editando, añadir capítulos, borrarlos o editar los flags envueltos en el capítulo seleccionado.

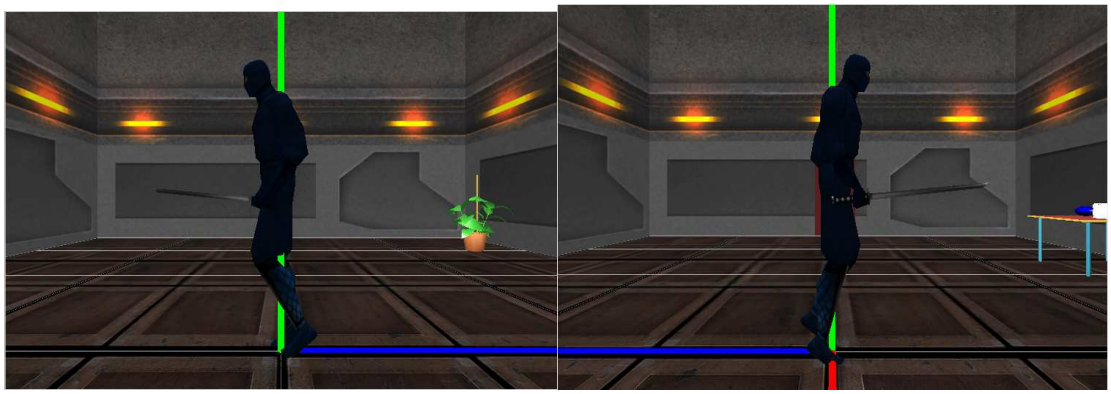
Menú "View"

En este menú podremos cambiar las cámaras de la ventana de previsualización. De esta manera, eligiendo "Zoom in" o "Zoom out" aumentamos o disminuimos el zoom. Además tenemos dos cámaras por defecto para cada eje. Así "Far (z axis)" y "Near (z axis)" nos dan una visión desde detrás o delante del eje Z:



De igual manera, "Top (y axis)" y "Bottom (y axis)" nos dan una visión desde arriba o abajo del eje Y:

Por último, "Left (x axis)" y "Right (x axis)" nos dan una visión a ambos lados del eje X, tanto a la izquierda como a la derecha.



Menú “Show”

Este menú nos dará la posibilidad de mostar o no determinados elementos de las escenas. Así, podremos activar o desactivar (para que sean mostrados o no) el techo, los muros y el suelo para cuartos, el cielo y el terreno para entornos abiertos, y en ambos, el jugador, los objetos y los personajes no principales.

Las opciones “model” y “extension” hacen referencia a los escenarios basados en un modelo 3D. Cuando “model” está activado dicho modelo se mostrará en la pantalla y análogamente con los cuatro límites (X max, X min, Z max, Z min) que establecen la zona en la que podrá moverse el jugador, siempre que se haya activado esta opción.

B. Files and folders of the project

B.1. Brief description of the project files and folders

In this section we list all the files and folder that form the project and their function or content. Let's start with the folders:

- *adventures* - It keeps the two games we have developed with <e-Adventure3D>: the game we explain in the chapter 'Caso de estudio' and the demo Tech.
- *bin* – Contains all the java classes compiled.
- Folder *defaulassets* – The default art assets used in the editor tool.
- *doc* – Contains this report and some files related with the documentation.
- *gui* – This folder stores images (such as the hud icons or the loading screens) and models (such as the door used in closed rooms) used in the engine.
- *img* – Images used in the editor tool, such as the icons for the buttons.
- *jars* – It stores the jar files used to execute the editor and the engine.
- *jme* – Contains all the necessary libraries to use JME.
- *rtf* – It keeps all the rtf files used in the editor.
- *saveData* – It is the default folder where the engine keeps the games saved.
- *src* – contains all the Java code.

Next we list the files:

- '*.classpath'* and '*.project'* files – these files save information of the project
- The DTD files: '*adaptation.dtd*', '*assessment.dtd*', *'descriptor3d.dtd*', *'eadventure3d.dtd'*. Those are used to validate each type of XML document in a EA3D file (see the chapter 'Characteristics of <e-Adventure3D>' for further details).
- 'Config.xml': it contains some configuration information for the editor tool such as the language or the files opened recently.
- *'editor.bat'* and *'engine.bat'*: files to execute the editor tool and the engine using microsoft windows.

- *'English.xml'*: the properties XML file with all the sentences written in the editor tool. (We described its function in the chapter 'Implementation')
- *'gamepadsConfig.xm'l* and '*movementsSettings.xml'* are described in the next section of this chapter.
- *'properties.cfg'* Properties file for the configuration of the Java Monkey Engine.

B.2. Properties files definition

During the first iteration the Java's properties layer has been used to provide customized settings for <e-Adventure3D>. Specifically talking, two are the properties files managed in <e-Adventure3D>: the game pads settings files (stores information about how buttons and sticks are distributed in the available game pads) and the movement settings files (stores data to adjust how the player moves), which are detailed in this section. Both files are marked-up using XML-based languages thanks to the utility provided by that properties layer.

It is important to remark that both files are not devised to allow users the modification of their contents. However, it is easy to manually modify the movement settings file in order to suit instructors and learners' tastes. On the other hand, game pad settings file is provided along with a configuration tool, currently under development, supplying a user-friendly interface to get game pads set up.

B.2.1.The Movement settings file

The movement settings file stores properties, following the XML language defined by the properties layer in Java, binding a key to a value. In this manner, the file could be structured in 3 sections, as depicted in the example given. The first section specifies the settings for movement when the scene is view from a static camera fixed in the same position all the time. Analogously the second section defines the settings for the other view supported by <e-Adventure3D>, the 3rd Person view. Both sections could be merged in one, establishing the same settings for the two view types, by simply removing the "-static" or "-3rdperson" suffix in each entry key. Finally a third section for the specification of those common settings for both static and 3rd person view. The next table describes which parameters are customizable and the syntax of language supporting that customization. Note that all the parameters are optional, so when not present a

default value is set.

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<!DOCTYPE properties SYSTEM "http://java.sun.com/dtd/properties.dtd">
<properties>
    <comment>Movement Settings</comment>
    <entry key="forwardSpeed-static">10.0</entry>
    <entry key="backwardSpeed-static">5.0</entry>
    <entry key="rotatingSpeed-static">0.7</entry>
    <entry key="fwAnimationSpeed-static">0.5</entry>
    <entry key="bwAnimationSpeed-static">0.5</entry>
    <entry key="rtAnimationSpeed-static">0.5</entry>

    <entry key="forwardSpeed-3rdPerson">50.0</entry>
    <entry key="backwardSpeed-3rdPerson">5.0</entry>
    <entry key="rotatingSpeed-3rdPerson">1.5</entry>
    <entry key="fwAnimationSpeed-3rdPerson">1</entry>
    <entry key="bwAnimationSpeed-3rdPerson">0.5</entry>
    <entry key="rtAnimationSpeed-3rdPerson">0.5</entry>

    <entry key="error">5</entry>
    <entry key="fixedBwAnimationSpeed"/>
    <entry key="fixedRtAnimationSpeed"/>
</properties>
```

Parameter	Syntax (Entry key)	Value (Entry value)	Description
Forward speed	forwardSpeed (-static | -3rdPerson)?	Speed when moving forward in pixels/s	Determines how many pixels is the player's avatar translated per second when moving forward
Backward speed	backwardSpeed (-static | -3rdPerson)?	Speed when moving backwards in pixels/s	Determines how many pixels is the player's avatar translated per second when moving backwards
Rotation speed	rotatingSpeed	Speed for rotation in degrees/s	Determines the angle in degrees the player must turn in a

	(-static | -3rdPerson)?		second when rotating (changing the direction of the movement)
Forward animation speed	fwAnimationSpeed (-static | -3rdPerson)?	Proportional factor. It is given in no units	Factor that when specified is used to multiply the default frame rate for the forward animation. For instance, if value=2 each frame is shown the half time, making the impression that the player speed has increased.
Backward animation speed	bwAnimationSpeed (-static | -3rdPerson)?		
Rotation animation speed	rtAnimationSpeed (-static|-3rdPerson)?		
Forward animation mode	fixedFwAnimationSpeed (-static | -3rdPerson)?	No value is required.	Forces the animation to behave digitally. By default all animations are analogue, that is are proportional to the intensity of the input source.
Backward animation mode	fixedBwAnimationSpeed (-static | -3rdPerson)?		
Rotation animation mode	fixedRtAnimationSpeed (-static | -3rdPerson)?		
Error	Error (-static | -3rdPerson)?	Percentage of error admitted for the input source.	Establishes the error of the game pads. That allows distinguishing whether an input signal has been produced by the user or by a malfunction of the device. In short, it gauges the sensibility of the input devices.

B.2.2 Game pad settings file

This file is analogous to the movement settings file in the manner it is managed and stored. It contains the information needed to map the game pads to the <e-Adventure3D> game pad, which is also described in this section.

<e-Adventure3D> games are supposed to be played using commercial game pads. Watching the market, a common game pad for PC games has usually 12 digital buttons and 2 analogue sticks (i.e. 4 axes). The next figure illustrates the

kind of game pad we are talking about, which design is probably the most widespread within the market, and the name has been given to each button and stick for their identification.

(1) Cross (2) Circle (3) Triangle (4) Square (5) R1 (6) R2

(9) Start(10 Select (11) Left stick(12) Right stick (7) L1 (8) L2

Note that in order to distinguish the two axes which conform each stick they have been called X and Y.

The <e-Adventure3D> game pad is used to abstract how JME names each button and axis for the different devices, which defer from the others depending on the trademark that produces them. Therefore the properties file stores for each game pad which has already been configured which button/axis match to the ones aforementioned.

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<!DOCTYPE properties SYSTEM "http://java.sun.com/dtd/properties.dtd">
<properties>
    <comment>Joysticks configuration for eAdventure3d</comment>
    <entry key="gamepads">Logitech Dual Action USB\USB Joystick     \Logitech Dual Action</entry>

    <entry key="gamepad(Logitech Dual Action)">4\12</entry>
    <entry key="gamepad(Logitech Dual Action)\axes">0\1\2\3</entry>
    <entry key="gamepad(Logitech Dual Action)\axis(0)">righty\right\y\1.0\-1.0</entry>
    <entry key="gamepad(Logitech Dual Action)\axis(1)">rightx\right\x\-1.0\1.0</entry>
    <entry key="gamepad(Logitech Dual Action)\axis(2)">lefty\left\y\1.0\-1.0</entry>
    <entry key="gamepad(Logitech Dual Action)\axis(3)">leftx\left\x\-1.0\1.0</entry>

    <entry key="gamepad(Logitech Dual Action)\buttons">0\1\2\3\4\5\6\7\8\9\11</entry>
    <entry key="gamepad(Logitech Dual Action)\button(0)">square</entry>
    <entry key="gamepad(Logitech Dual Action)\button(1)">cross</entry>
    <entry key="gamepad(Logitech Dual Action)\button(2)">circle</entry>
    <entry key="gamepad(Logitech Dual Action)\button(3)">triangle</entry>
    <entry key="gamepad(Logitech Dual Action)\button(4)">L1</entry>
    <entry key="gamepad(Logitech Dual Action)\button(5)">R1</entry>
    <entry key="gamepad(Logitech Dual Action)\button(6)">L2</entry>
    <entry key="gamepad(Logitech Dual Action)\button(7)">R2</entry>
    <entry key="gamepad(Logitech Dual Action)\button(8)">start</entry>
    <entry key="gamepad(Logitech Dual Action)\button(9)">select</entry>
    <entry key="gamepad(Logitech Dual Action)\button(11)">R3</entry>
</properties>
```

Bibliography and References

[1] Moreno-Ger, P. (2007). A Documental Approach to the Creation and Integration of Digital Videogames in Virtual Learning Environments. *Ph.D. thesis, Universidad Complutense de Madrid*.

[2] http://e-adventure.e-ucm.es <e-Adventure> web site. Checked on the 13th of June of 2008.

[3] Gee, J.P. (2003). *What videogames have to teach us about learning and literacy*. New York: Palgrave McMillan.

[4] Fernández-Manjón, B., Moreno-Ger, P., Sierra, J. L., Martínez-Ortiz, I. (2007). *Uso de estándares aplicados a TIC en Educación*. Report #16 CNICE (National Center for Educational Information and Communication). NIPO 651-06-344-7, 651-06-345-2. Available at

http://ares.cnice.mec.es/informes/16/contenido/indice.htm. 2007

[5] Ondrejka, C., Cook, J., Conklin, M. (2005). How user content changes everything. *Games Learning and Society Conference*. June 23, 2005.

[6] Conklin, M. (2007). 101 Uses for Second Life in the Collage Classroom.

http://facstaff.elon.edu/mconklin/pubs/glshandout.pdf Checked on the 13th of June of 2008.

[7] http://secondlifegrid.net/programs/education Web site about education in second life. Checked on the 13th of June of 2008.

[8] http://secondlife.com/ Second life web site. Checked on the 13th of June of 2008.

[9] http://www.simteach.com/wiki/index.php?title=Second_Life_Education_Wiki

On-line list of education organizations in second life. Checked on the 13th of June of 2008.

[10] http://www.gamevillage.nl/review/topic/145212-1.html On-line list of game authoring tools. Checked on the 13th of June of 2008.

[11] http://www.fpscreator.com/ FPS Creator tool web site. Checked on the 13th of June of 2008.

[12] http://3das.noeska.com/default.aspx 3D Adventure Studio web site. Checked on the 13th of June of 2008.

[13] http://skymatter.thegamecreators.com/ Tool for the creation of sky boxes. Checked on the 13th of June of 2008.

[14] http://t3dgm.thegamecreators.com/ 3D Game Maker web site. Checked on the 13th of June of 2008.

[15] Aldrich, C. (2004). *Simulations and the Future of Learning: an Innovative (and Perhaps Revolutionary) Approach to e-Learning*. San Francisco, CA: Pfeiffer.

[16] Aldrich, C. (2005). *Learning by Doing: A Comprehensive Guide to Simulations, Computer Games and Pedagogy in e-Learning and Other Educational Experiences*. San Francisco, CA: Pfeiffer.

[17] Malone, T. (1981). Toward a theory of intrinsically motivating instruction. *Cognitive Science, 5, 333–369*.

[18] Malone, T. (1982). What makes computer games fun? *SIGSOC Bulletin, 13(2-3), 143*.

[19] Prensky, M. (2001). Digital Game Based Learning. New York: McGraw-Hill.

[20] Squire, K., y Barab, S. (2004). Replaying history: Engaging urban underserved students in learning world history through computer simulation games. En Y. B. Kafai,W. A. Sandoval, N. Enyedi, A.Nixon, y F. Herrera (Eds.) Sixth International Conference of the Learning Sciences, (pp. 505–512). Santa Monica, United States: Lawrence Erlbaum Associates.

[21] Starr, P. (1994). Seductions of Sim. The American Prospect, 2(17).

[22] Kolson, K. (1996). The politics of SimCity. Political Science and Politics, 29(1), 43–46.

[23] Academic ADL Co-Lab (2004). Outbreak Quest: A 90-day game development initiative. Available on-line on:

http://www.academiccolab.org/resources/documents/OutbreakQuest.pdf. Checked on the 13th of June of 2008.

[24] Overmars, M. (2004). Teaching computer science through game design. IEEE Computer, 37(4), 81–83.

[25] Robertson, J., y Good, J. (2005). Story creation in virtual game worlds. Communications of the ACM, 48(1), 61–65.

[26] Amory, A., Naicker, K., Vincent, J., y Adams, C. (1999). The use of computer games as an educational tool: Identification of appropriate game types and game elements. British Journal of Educational Technology, 30(4), 311–321.

[27] Ju, E., y Wagner, C. (1997). Personal computer adventure games: Their structure, principles and applicability for training. The Database for

Advances in Information Systems, 28(2), 78–92.

[28] Van Eck, R. (2007). Building artificially intelligent learning games. En D. Gibson, C. Aldrich, y M. Prensky (Eds.) Games and Simulations in Online Learning: Research and Development Frameworks. Hershey, PA: Information Science Publishing.

[29] http://www.yoyogames.com/gamemaker/ Game Maker took web site. Checked on 13th of June of 2008.

[30] http://www.alice.org/ Alice tool web site. Checked on 13th of June of 2008.

[31] http://www.immersiveeducation.com/MissionMaker/ Mission Maker web site. Checked on 13th of June of 2008.

[32] http://www.adventuregamestudio.co.uk/ Adventure Game Studio web site. Checked on 13th of June of 2008.

[33] http://www.adventuremaker.com/ Adventure Maker web site. Checked on 13th of June of 2008.

[34] Kirriemur, J., y McFarlane, A. (2004). *Literature review in games and learning*. Tech. Rep. 8, NESTA Futurelab.

[35] Goldberg, M., y Salari, S. (1997). *An update on WebCT (World-Wide-Web Course Tools) - A tool for the creation of sophisticated web-based learning environments*. In NAUWeb 97 - Current Practices in Web-Based Course Development. Flagstaff, Arizona (United States).

[36] Parker, A. (2003). *Identifying predictors of academic persistence in distance education*. Journal of the United States Distance Learning Association, 17(1), 55–62.

[37] Levy, Y. (2007). *Comparing dropouts and persistence in e-learning courses*. Computers & Education, 48(2), 185–204.

[38] Oz, E., y White, L. (1993). *Multimedia for better training*. Journal of Systems Management, 44(5), 34–38.

[39] Martinez-Ortiz, I., Moreno-Ger, P., Sierra, J. L., and Fernández-Manjón, B., *Production and Deployment of Educational Videogames as Assessable Learning Objects*, in First European Conference on Technology Enhanced Learning (ECTEL 2006), Crete, Greece, 2006, pp. 316-330.

[40] https://www.ucm.es/campusvirtual/CVUCM/index.php. Web site of the WebCT virtual campus of the Complutense University, Madrid.

[41] Michael, D. and Chen, S. (2006). *Serious Games: Games that Educate, Train, and Inform*. Boston, MA: Thomson.

[42] Sommerville, I. (2007). *Software Engineering*. Adison Wesley.

[43] Srevens, P., Pooley, R. (2007). *Utilización de UML en ingeniería del software con objetos y componentes*. Adison Wesley.

[44] Pressman, R. S. (2005). *Ingeniería del Software*. McGraw-Hill Interamericana.

[45] Moreno-Ger, P., Burgos, D., Sierra, J. L., Fernández-Manjón, B. (2007). *A Game-Based Adaptive Unit of Learning with IMS Learning Design and <e-Adventure>*. Proceedings of the Second European Conference on Technology Enhanced Learning (EC-TEL 2007), Crete, Greece. Lecture Notes in Computer Science 4753, 247–261 (Springer). 2007

[46] Moreno-Ger, P., Blesius, C., Currier, P., Sierra, J. L., Fernández-Manjón, B. (2007). *Rapid Development of Game-like Interactive Simulations for Learning Clinical Procedures*. In Proceedings of the Fifth International Game Design and Technology Workshop and Conference (GDTW2007), pages 17-25. Liverpool, UK. (Received the Best Paper Award from the Program Committee). 2007

[47] Moreno-Ger, P., Sancho, P., Martínez-Ortiz, I., Sierra, J. L., Fernández-Manjón, B. (2007). *Adaptive Units of Learning and Educational Videogames*. Journal of Interactive Media in Education 2007/05. [http://jime.open.ac.uk/2007/05]. 2007